オートマトン 言語理論 計算論 I

［第2版］

J.ホップクロフト／R.モトワニ／J.ウルマン＝共著
野崎昭弘／高橋正子／町田 元／山崎秀記＝共訳

INFORMATION & COMPUTING

サイエンス社

Information & Computing—3

オートマトン 言語理論 計算論 I
［第2版］

J.ホップクロフト／R.モトワニ／J.ウルマン＝共著
野崎昭弘／高橋正子／町田　元／山崎秀記＝共訳

サイエンス社

サイエンス社のホームページのご案内
http://www.saiensu.co.jp
ご意見・ご要望は　rikei@saiensu.co.jp　まで．

訳者まえがき

本書は，原著者の序文にも記されている通り，1969年にホップクロフトとウルマンの共著として出版された『言語理論とオートマトン』(サイエンス社，1971年) と，1979年に出版された『オートマトン・言語理論・計算論／I・II』(サイエンス社，1984年・86年) の全面改訂版である．学部の講義に合うように，深さを追求するよりは「基礎的な部分の直観的な理解を確実にする」ことがめざされていて，前著よりずっと説明が詳しくなっている．その分「風格が落ちた」とか「自分で考えるところが少なくなった」という批判もあろうが，これでもまだ「むずかしい」と感じる読者はおられるだろうから，これまでの著書に代わる，新しい教科書の一つの標準になることと予想される．

翻訳の分担は以下の通りである．

1, 5, 6章と序文	… 野崎昭弘
2, 3章	… 高橋正子
4, 10, 11章	… 町田元
7, 8, 9章	… 山崎秀記

なお分量の関係で，第1章から7章までをI巻，そのあとをII巻として出版することになった．訳語については，前著にあるものは基本的にそれに従い，新しいものは標準的な情報科学辞典などを参照して，なるべく一般的な訳語を使うように努めた．しかしそれでもいくつかの用語について，訳者の間で意見が分かれる場合もあった．たとえば本文中のinductionはもちろん「(数学的) 帰納法」でよいが，証明の中に現れるBasis ($n=1$の場合) とInduction (nから$n+1$へ) については,「帰納的ステップ」,「帰納法のステップ」などを含む意見が対立し，結局野崎の判断（というよりは趣味で)「基礎」,「帰納」という短い用語を当てることにした（本文中のinductive stepは「帰納的ステップ」にしたので,「帰納」はその省略形とお考えいただきたい)．

本訳書によって，これからも日本の情報科学の土台がしっかり維持されることを，訳者一同期待している．しかし何しろ大部の書物であるから，思わぬ見落としや不適切な翻訳が残っているかも知れない．お気づきの点は，どしどし

ご教示いただけると幸いである．

編集の段階で，サイエンス社編集部の田島伸彦さん，鈴木綾子さんには大変お世話になった．特に鈴木さんは原稿と校正刷りを綿密にチェックされ，訳者の見落とした不統一・訳し洩れ・誤字などを数多く指摘して下さった．この場を借りて深くお礼を申し上げたい．

2003 年 2 月 18 日

野崎昭弘
高橋正子
町田　元
山崎秀記

序　文

1979 年に出た本書の前身では，序文の中でホップクロフトとウルマンはオートマトンの話題が，彼らの最初の本が出た 1969 年頃の状況にくらべて，爆発的に拡がったという事実に驚嘆している．実際，1979 年の本は前の著書になかった多くのトピックを含んでいて，大きさは約 2 倍に膨らんでいた．その 1979 年版と比べると，本書は 1970 年代の自動車のように,「外から見ると大きくて，中は小さく」なっている．これは逆戻りのように聞こえるかも知れないが，著者たちはいくつかの理由でこの変化を歓迎している．

まず 1979 年には，オートマトンと言語の理論は活発な研究領域であった．あの本の目的は数学的な傾向の学生がその分野で新しい貢献をするのを励ますことであった．今日，オートマトン理論を直接めざす研究は（応用とは対照的に）僅かで，1979 年版の簡潔で，高度に数学的なトーンを保つ理由はほとんどない．

第 2 に，オートマトンと言語の理論の役割は，過去 20 年間で変わった．1979 年には，オートマトンはおもに大学院レベルの科目で，我々は読者として進んだ大学院生を想定していた——後半の章を読む学生は，特にそうである．今では，この科目は学部カリキュラムの一つの基本科目である．そこで本書の内容は，学生の予備知識を少なくし，基礎的な事柄の説明を増やし，前著より詳しい議論を書かねばならなかった．

第 3 に環境の変化がある．CS（コンピュータ科学）が過去 20 年間に予想を超えて成長を遂げた．1979 年にはカリキュラムを「これから次の技術革新を越えて生き残る」と確信できる材料で満たすのは容易なことではなかったのに，今では非常に多くの話題・技法が，学部カリキュラムの限られた時間の中でせめぎあっている．

第 4 に，CS はより職業的な科目になり，多くの学生に厳しい実利主義が生まれている．我々はオートマトン理論の諸相が多くの新しい教科で本質的な道具であると今でも信じているし，学生がすぐにお金になる技術だけを習いたいとどれほど強く望んだとしても，オートマトン理論の標準的なコースに含まれている理論的で視野を拡げる練習問題は今でも価値を失っていない，と信じて

いる．しかし，コンピュータ科学の学生が選べる科目リストの中に確実に位置を占め続けるためには，数学と並んで応用を強調する必要がある，と思われる．そのため，前著のいくつかの抽象的な話題を現在そのアイデアがどのように使われているかの例示に置き換えた．オートマトンと言語の理論のコンパイラへの応用はよく知られていてふつうコンパイラのコースでカバーされるが，他にもプロトコルを検証するモデル検査アルゴリズム，文脈自由言語を模型とするドキュメント記述言語など，新しい応用がたくさんある．

本書で拡大と縮小が同時進行した最後の要因は，ドン・クヌースとレス・ランポートが開発した植字システム TeX と LaTeX を利用できたことである．特に後者のお陰で，植字の「開かれた」方式が可能になり，本書が大きく，しかし読みやすくなった．我々はお二人の努力に感謝している．

この本の使用法

本書は 3 年生以上の半年コース，あるいはその半分のコースで使うのに適している．スタンフォードでは，オートマトンと言語理論の中のコース CS153 でこの本書の素稿を使ったが，これは 4 分の 1 年のコースで，Rajeev と Jeff の 2 人が教えた．時間が限られていたので，第 11 章は扱わず，こみいった多項式時間リダクションなど，後ろの方の材料のいくつかも省かれた．本書のウェブ・サイト（あとに示す）には，CS154 のための何回かの講義ノートとシラバスがのっている．

何年か前に，我々はスタンフォードに来た大学院生の多くが，オートマトン理論は習っているが「実際的に解けない」可能性の理論を習っていないことに気がついた．スタンフォードの教員はコンピュータ科学者なら誰でも「NP 完全だととても時間がかかる」程度では困るので，それ以上の理解を得るためにはその理論は欠かせないと考えているため，もう一つのコース CS154N が用意されている．そのコースでは学生は第 8，9，10 章を読めばよい．実際には CS154N のための予備知識としては，CS154 の最後の 3 分の 1 で，これらの章が役立つ．今日でも，各学期ごとにこの選択科目を活用する学生がいる．余分な努力はごくわずかですむので，この選択はおすすめしたい．

予 備 知 識

本書を利用する最もよい方法は，グラフ，木，論理，および証明技法などを含

む離散数学のコースを取っておくことである．プログラミングのコースや，一般的なデータ構造，再帰，コンパイラなどシステムの主要な構成要素についても学んでいることを仮定している．これらの基礎科目は，1〜2 年生用の標準的な CS プログラムで学べる．

演 習 問 題

本書はほとんどすべての節にいくつかずつ，幅広い練習問題をのせている．むずかしい問題（あるいはむずかしい部分）には感嘆符 **!** がついている．最高にむずかしい問題には二重の感嘆符 **!!** がついている．

星印 * がついた問題（あるいはその一部分）もある．そういう問題には，ウェブ・ページを通して解答を見られるように努力している．それらの解答は公開され，自習のために利用できる．場合によっては，ある問題 B を解くのに別の問題 A の解答の変形あるいは応用が必要になることに注意してほしい．問題 A のある部分が解けたなら，B の対応する部分も解けると思ってよい．

WWW によるサポート

本書のホームページは次の通り．

`http://www-db.stanford.edu/~ullman/ialc.html`

ここには星印つきの問題の解答，気がついた誤りの訂正，参考資料が掲載されている．我々は CS154 を教えるたびに，その講義ノートを（宿題・解答・試験問題も）利用できるようにしたいと思っている．

謝 辞

「証明をどう書くか」という Craig Silverstein のプリントは，第 1 章の一部の材料として役だった．本書の素稿について，コメントと誤りの訂正が次の人々から寄せられた：Zoe Abrams, George Candea, Haowen Chen, Byong-Gun Chun, Jeffrey Shallit, Bret Taylor, Jason Townsend, Erik Uzureau
これらの人々に深く感謝する．残った誤りはもちろん我々の責任である．

ニューヨーク・イタカ，カリフォルニア・スタンフォード

2000 年 9 月　　　J. E. H.　R. M.　J. D. U.

目　　次

1　オートマトン：奇妙な，しかし適切な方法

2 有限オートマトン

3 正則表現と正則言語

4 正則言語の性質

7　文脈自由言語の性質

[第 II 巻]

1 オートマトン：奇妙な，しかし適切な方法

オートマトン理論とは，抽象的な計算装置，あるいは「機械」の研究のことである．コンピュータがなかった 1930 年代に，A. テューリング（A. Turing）は現代のコンピュータと，何が計算できるかについてまったく同じ能力を持つ抽象的な機械を考察した．テューリングの目標は，計算機械が「できること」と「できないこと」の間の境界を明確に記述することであった．その結論は，彼の抽象的な「テューリング機械」だけでなく，今ある機械についてもあてはまる．

1940 年代と 50 年代には何人かの研究者たちが，現在「有限オートマトン」と呼ばれている，より簡単な型の機械を研究した．それらのオートマトンは，もともと脳の機能のモデルとして提案されたものであったが，我々が 1.1 節で紹介する他のいろいろな目的に非常に有効であることがわかった．1950 年代の後半には，言語学者 N. チョムスキー（N. Chomsky）が形式的な「文法」の研究を始めた．その文法は，厳密には機械ではないが，抽象的なオートマトンと密接な関係があり，現在ソフトウェアのある重要な構成要素，たとえばコンパイラの一部分の基礎として役立っている．

1969 年に，クック（S. Cook）はテューリングの「計算できること，できないこと」の研究を発展させた：彼は「コンピュータで効率よく解ける問題群」と「原理的には解けるが実際問題として非常に時間がかかるため，ごく小規模な場合しかコンピュータが役に立たないような問題群」とを分離することに成功した．あとの問題群は「手に負えない」とか「NP 困難」と呼ばれている．計算速度の指数関数的な向上（ムーア（E. Moore）の法則）が続いたとしても，大きな規模の「手に負えない問題」が解けるようになるとは考えられない．

これらの理論的な発展はすべて，コンピュータ科学者が今行っていることに直接関係している．ある概念は，有限オートマトンとかある種の形式文法のように，重要なソフトウェアの設計と開発に使われている．他の概念は，テューリング機械のように，ソフトウェアに何ができるかを理解する助けになってい

る．特に,「手に負えない問題」の理論は，ある問題が（手に負えない問題ではないから）真正面から取り組みプログラムを書けるか，それとも（手に負えない問題なので）近似解を探す・発見法を利用する・所要時間を抑える何かの手段を考えるなどの道を探るべきか，を判断できるようにしてくれる．

この序章では，オートマトン理論とは何かを述べ，どんな役に立つのかをおおまかに展望することから始める．この章の主要部分で，証明技法と発見的証明法を一通り紹介する．演繹的証明，命題の言い換え，背理法，帰納法による証明などの重要な概念が含まれている．最後の節で，オートマトン理論のいたるところで使われる，アルファベット，文字列，言語の概念が導入される．

1.1 オートマトン理論をなぜ学ぶのか?

オートマトンと計算量の理論がどうしてコンピュータ科学の重要な核心部分を占めているかには，いくつかの理由がある．この節は主要な動機づけを紹介し，この本で取り扱う主要な話題の概略を説明する．

1.1.1 有限オートマトンとは

有限オートマトンは多くの重要な型のハードウェアとソフトウェアの便利なモデルである．この概念を活用するのは第2章からになるが，ここでは最も重要な種類の応用を列挙しておこう．

(1) デジタル回路の動作を設計しチェックするためのソフトウェア

(2) 典型的なコンパイラの字句解析，すなわち入力列を「名前」や「予約語」，句読点などの論理的な単位に分解する，ソフトウェアの構成要素

(3) ウェブ・ページの集まりのような大量のテキストを走査して，単語や語句などそのパターンの有無を探索するソフトウェア

(4) 通信プロトコルや安全な情報交換プロトコルなど，いくつか有限個の状態を取り得るあらゆる種類のシステムの検査を行うソフトウェア

いろいろな種類のオートマトンの正確な定義はすぐあとで述べるが，有限オートマトンとはどんなもので何をするのかを描く直観的な導入から始めよう．上に列挙されたような多くのシステムあるいは構成要素は，いつも有限個の「状態」の内の一つを取ると考えられる.「状態」の目的は，システムの歴史の必要な部分を記憶することである．状態は有限個しかないので，歴史のすべてを記憶することは普通できない．だからシステムは大事なことを記憶し，そうでな

いことは忘れるように，注意深く設計しなければならない．状態が有限個しかないことの利点は，システムを有限の資源で実現できることである——たとえば回路としてハードウェアの中に，限られた量のデータを参照するかまたは今実行中の場所に基づいて決定を行う単純なプログラムの形で，実現できる．

例 1.1　おそらく最も簡単で意味のある有限オートマトンは，オンとオフの切り替えスイッチであろう．この装置は今「オン」の状態にあるか「オフ」の状態にあるかを記憶し，利用者がボタンを押すことによって状態が変わる．ボタンの効果はスイッチの状態によって違うので，スイッチがオフのときにボタンを押せばオンの状態に変わり，スイッチがオンの状態のときにボタンを押せばオフの状態に変わる．

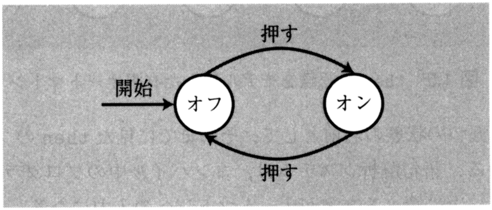


図 1.1　オン/オフ・スイッチをモデル化した有限オートマトン

スイッチを表す有限オートマトン・モデルを，図 1.1 に示した．状態はいつも，円で表される：この例では，状態の名前「オン」と「オフ」を書き込んでおいた．状態の間の矢印には“入力”の名前がつけられている——入力とは，外界からシステムへの影響のことである．ここではどちらの入力にも（ボタンを）「押す」という名前が与えられている．二つの矢印の意図は，どの状態にあっても「押す」入力が与えられたときにどの状態に移るかを示すことである．

状態のうちの一つは「初期状態」と呼ばれ，システムは最初その状態にある．この例では，オフ状態がそれで,「開始」と書かれた，その状態に向かう矢印で示すのがふつうである．いくつかの「最終状態」あるいは「受理状態」が指定されることもある．一連の入力によってシステムが受理状態の一つに到達するなら，その入力列はある意味で「よい」列である．たとえば図 1.1 では，そのスイッチによって制御されている装置が働くのはスイッチがオンのときである

から，状態オンを「受理状態」と考えることができる．ふつう受理状態は2重の円で示されるが，図1.1ではそのような指定はしていない．▨

例1.2 状態によって記憶されていることは，オン/オフ・スイッチよりずっと複雑なことがある．図1.2は，語句解析の一部分となり得る有限オートマトンを示している．このオートマトンの任務はキーワード then の認識である．そのため単語 then のどこまで進んだか，その位置を表す五つの状態が必要である．それらの位置は，その単語の（広義）プレフィックス，すなわち空列（何も見ていない状態）から完全な単語までの先頭部分に対応している．

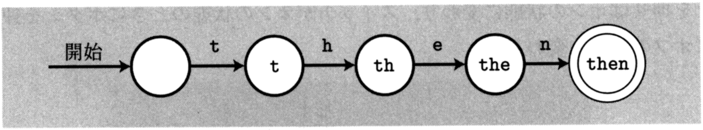


図1.2　then の認識をモデル化した有限オートマトン

図1.2では五つの状態の名前として，それまでに見た then のプレフィックスが使われている．語句解析システムは，コンパイル中のプログラムの文字を一つずつ調べて，次に調べる文字がオートマトンへの入力になる，と考えられる．初期状態は空列に対応し，どの状態も then の中の次の文字によって，その次に大きいプレフィックスに対応する状態へと移行する．名前が then の状態に到達するのは，入力された語の綴りが then に一致したときである．このオートマトンの任務は then の認識であるから，then と名付けられた状態がただ一つの受理状態である．▨

1.1.2 構造的表現

オートマトンと見かけは違うが，オートマトンとその応用の研究に重要な役割を果たす概念が二つある．

(1) **文法**は，再帰構造を持つデータを扱うソフトウェアの設計に便利なモデルである．最も有名な例は，構文解析を行う“パーサー”で，これは典型的なプログラミング言語にある再帰的な入れ子構造，たとえば数式や論理式，等々を処理する，コンパイラの構成要素である．たとえば，$E \Rightarrow E + E$ のような文法的規則は，任意の二つの式をプラス記号でつないでまた式を

作れる，ということを示している．これは現実のプログラミング言語で式を形成する標準的な方法である．我々は第 5 章で文脈自由言語（これがよく使われる文法の名前である）を導入する．

(2) 正則表現も，データの構造，特に文字列のデータ構造の表示する．第 3 章で述べるように，正則表現で記述できる文字列のパターンは，有限オートマトンで記述できるものと正確に一致する．この表現の見かけは文法とまったく異なり，ここでは簡単な例を示すだけにしておこう．UNIX ふうの正則表現

```
'[A-Z][a-z]*[ ][A-Z][A-Z]'
```

は「大文字で始まる語のあとに一つの空白と二つの大文字が続く」文字列を表している．この表現は，テキストの中の「市と州」，たとえば `Ithaca NY` のようなパターンを表している．市の名前が複数語で表される `Palo Alto CA` のような場合は洩れてしまうが，それもさらに複雑な表現を使えば含められる：

```
'([A-Z][a-z]*[ ])*[A-Z][A-Z]'
```

このような表現を理解するには，`[A-Z]` が大文字の A から Z までの範囲（つまり一つの任意の大文字）を表し，`[ ]` が 1 個の空白を表すことを知らないといけない．また記号 `*` は，その前の表現を「何回でも繰り返す」ことを表している．括弧 `()` は表現のグループ分けに使われ，記述されているテキストの文字を表すものではない．

1.1.3 オートマトンと計算量

オートマトンは計算の限界の研究に本質的である．最初に述べたように，重要な問題が二つある．

(1) コンピュータは一体何ができるのだろうか？ キーワードは「決定可能性」で，コンピュータに解ける問題は「決定可能」と呼ばれる．これについては第 9 章で扱う．

(2) コンピュータが効率よくできるのは何だろうか？ キーワードは「手に負えるか否か」で，コンピュータに効率よく解ける問題（入力の大きさに対して何かゆっくりと大きくなる関数で所要時間がおさえられる問題）は

「手に負える」といわれる．多項式は「ゆっくりと大きくなる」とみなし，どの多項式よりも急速に大きくなる関数は「成長が早すぎる」とみなされることが多い．この問題については第10章で学ぶ．

1.2　形式的な証明とは

もしあなたが1990年代以前に高校で平面幾何を習っていたなら，おそらく「演繹的証明」，すなわち「主張の正しさを，段階を細かく分けて，論理にしたがって示す」ことをかなりの程度やったに違いない．幾何には実用的な面もある（部屋に敷く正しい大きさの絨毯を買うには，長方形の面積を計算する公式を知らないといけない）が，形式的な証明の方法論を学ぶことも，高校で数学のこの分野を教えることの重要性において，それに劣らない．

アメリカでは1990年代に，証明を個人的な感情にかかわる事柄として教えることが盛んになった．たしかに使いたい文が正しいと感じることはよいことであるが，証明の重要な技法は教えられなくなった．しかし証明はすべてのコンピュータ科学者が理解しなければならないものである．コンピュータ科学者の中には，次のような極端な主張をする人もいる：プログラムを書くときには，それと並行してそのプログラムの正しさの形式的証明を進めなければならない．我々はそれが生産的であるとは思わない．一方,「プログラミングの領域では，証明などまったく要らない」という人もいる．彼らの陣営では「プログラムの正しさに確信が持てなければ，走らせて見よ」というスローガンがはやっている．

我々の立場はこの両極端の中間にある．プログラムの検査はたしかに重要である．しかしすべての入力を検証することは不可能なので，検査ではある程度の保証しかできない．さらに，プログラムが技巧的な再帰とか反復を含む複雑なものであると，繰り返す範囲や再帰的な呼び出しで何が行われるかよく理解していないと，正しいプログラムを書く望みはほとんどない．検査で「正しくない」とわかっても，どのように直せばよいかが大問題である．

反復や再帰を正しく記述するには，帰納的な仮定を定式化するとよい．そしてその仮定が反復あるいは再帰にうまくあてはまることを形式的あるいは直観的に証明するのは，とても役に立つことである．正しいプログラムの働きを理解するこの作業は，定理を帰納法で証明する作業と本質的に変わらない．そこ

でオートマトン理論のコースは，ある種のソフトウェアに役立つモデルを提供するだけでなく，形式的証明の方法論をも扱う習慣ができた．たぶんコンピュータ科学のどの基礎科目よりも，オートマトン理論は自然な証明にも技巧的な証明にも向いているので，**演繹的**なもの（保証された推論の列）でも**帰納的**なもの（変数つきの文を，それ自身のより小さな変数を持つ場合から帰納的に証明する）でも，実例が多い．

1.2.1 演繹的証明

上に述べた通り，演繹的証明とは文の列であって，ある出発点となる「**前提**」あるいは「**所与の文**」(複数個でもよい）から正しい「**結論**」を導くものである．どの文も，所与の事実・それ以前の（すでに証明された）文，またはそれらの組み合わせから，正しい推論の法則にしたがって導かれたものでなければならない．

前提は，それが含む変数の値によって，真にも偽にもなる．前提は，いくつかの文を論理的な積（「でしかも」，AND）でつないだものであることが多い．そのような場合，それらの文のひとつひとつを「前提」と呼ぶ．＜逆に，いくつか複数個の前提が使われているとき，それらを AND でつないで一つの前提とみなすこともできる．＞

仮定 H から結論 C が証明されたとき，文「もし H ならば C」を**定理**と呼ぶ.「C は H から**導かれる**」ともいう．次に「もし H ならば C」型の定理の例を示そう．

定理 1.3　もし $x \geq 4$ ならば，$2^x \geq x^2$ ▨

この定理の正しさを形式的に証明するには数学的帰納法が必要で，例 1.17 であらためて扱うが，直観的に理解するのはそうむずかしくない．まず前提 H が $x \geq 4$ であることに注意しよう．この前提は変数 x を含んでいるので，真とも偽ともいえない．というよりは，その正しさは変数 x の値によってきまる：$x = 6$ ならば H は真であり，$x = 2$ ならば偽である．

同じように，結論 C は $2^x \geq x^2$ で，これも変数 x を含んでいるから，x のある値に対しては真，他の場合には偽となる．たとえば $x = 3$ だと $2^3 = 8$ で，これは $3^2 = 9$ より小さいから，C は偽となる．一方 $x = 4$ ならば，$2^4 = 4^2 = 16$ であるから，C は真である．$x = 5$ のときも，$2^5 = 32$ は $5^2 = 25$ より大きい

ので，C はやはり真である．

結論 $2^x \geq x^2$ が $x \geq 4$ である場合にはいつでも正しいことは，おそらくもう明らかであろう．すでに $x = 4$ のときに正しいことはすでに見た通りである．x が4より大きくなると，左辺の 2^x は x が1ふえるたびに2倍になる．しかし右辺の x^2 は，$((x+1)/x)^2$ という割合でしか大きくならない．もし $x \geq 4$ であれば，比 $(x+1)/x$ は1.25を越えず，したがって $((x+1)/x)^2$ が1.5625を越えることはない．そして $1.5625 < 2$ であるから，x が4より1ずつ大きくなって行くとき，左辺の 2^x は右辺の x^2 よりさらに大きくなる．このように，不等式 $2^x \geq x^2$ が成り立つとすでにわかっている $x = 4$ から始めれば，x の値をいくら大きくしても，不等式はそのまま成り立つ．

これで定理1.3の直観的な，しかし正確な証明ができた．あとで「帰納的」証明を学んでから，例1.17でこの証明をもっと厳密に行う．

定理1.3は，他の興味ある定理と同様，無限個の事実に関連している：この場合は「すべての整数 x について，もし $x \geq 4$ ならば $2^x \geq x^2$ である」ということである．実は x が整数と仮定しなくてもよいが，上の証明では x を4から1ずつ大きくしているので，x が整数である場合しか扱っていない．

定理1.3は，他の定理を導くのに役立つ．次の例は，定理1.3を利用する簡単な定理の純粋に演繹的証明を示している．

定理 1.4　もし x が四つの正整数の平方の和で表されるならば，$2^x \geq x^2$ である．

証明　証明のアイデアを直観的にいうと，次のようになる．もし前提がある x について正しければ，すなわち x が四つの正整数の平方の和であれば，x は4以上である．だから定理1.3の前提が満たされ，この定理を信用する以上，その結論も同じ x について成り立つことがいえる．この推論は，段階に分けて述べられる：どの段階も証明すべき定理の前提であるか，その前提の一部分であるか，あるいはそれ以前の文（複数個でもよい）から導かれる．

ここで「導かれる」とは，もしある定理の前提が（証明の中の，すでに導かれた）前の文に一致するなら，その定理の結論は正しく，「次の文」として証明に書き加えてよいということである．この法則はよく「三段論法」（modus ponens）と呼ばれる——たとえば，もし H が正しく「もし H ならば C」が正しいとわかっているなら，C も正しいと結論してよい，ということである．前の複数個の文から新しい文を作り出す法則は他にもある．たとえばもし前の二つの文 P，Q から，「P でしかも Q」を導き

証明に書き加えることができる.

図 1.3 は定理 1.4 を証明するのに必要な文の列を示している. このような形に証明を記述することはふつうしないが, 証明とは文の非常に具体的なリストであり, 各文ごとにその根拠が示されているものである, と考える助けになる. ステップ (1) では, 定理の前提「x は四つの正整数の平方の和である」を繰り返している. ここでは四つの正整数を a, b, c, d で表しているが, このように「前提の中で使われているが名前のない量」に名前をつけておくと, 証明が書きやすくなることがよくある.

	文	根拠
(1)	$x = a^2 + b^2 + c^2 + d^2$	前提
(2)	$a \geq 1\,;\ b \geq 1\,;\ c \geq 1\,;\ d \geq 1$	前提
(3)	$a^2 \geq 1\,;\ b^2 \geq 1\,;\ c^2 \geq 1\,;\ d^2 \geq 1$	(2) と数の性質
(4)	$x \geq 4$	(1), (3) と数の性質
(5)	$2^x \geq x^2$	(4) と定理 1.3

図 1.3 定理 1.4 の形式的証明

ステップ (2) には, 定理の前提のもう一つの部分, つまり「平方される数は 1 以上」を書いた. 細かいことをいえば, ここには四つの整数についての独立の条件をまとめて書いている. ステップ (3) では, ある数が 1 以上であれば, その平方も 1 以上であることを利用している. そこで前の文 (2) と「数の性質」を根拠にしている. だからここでは, 読者が不等式の扱い方を知っていて,「もし $y \geq 1$ ならば $y^2 \geq 1$」のような簡単な文を知っているかまたは証明できることを仮定している.

ステップ (4) では前の文 (1) と (3) を使う. 文 (1) は x が四つの平方数の和であることを述べ, 文 (3) はどの平方数も 1 以上であるといっている. 再びよく知られた数の性質から, x は $1+1+1+1$ 以上, つまり 4 以上である.

最後のステップ (5) では文 (4) を使う. これは定理 1.3 の前提なので, 定理 1.3 を根拠として, その結論 (5) を導くことができる. これが定理 1.4 の結論であるから, 定理 1.4 の証明ができた. すなわち, 定理の前提から出発して, その結論を導き出すことができた. ▨

1.2.2 定義による還元

上の二つの定理では, 前提に整数, 和, 平方など, よく知られた言葉しか出てこなかった. しかし他の定理では, オートマトン理論の定理も含めて, 文中にいろいろな含みがあり明らかとはいえない術語が現れることがある. そこで有効な方法は：

- 何から手をつけていいかわからないときは, 前提の中のすべての術語を定

義にしたがって書き換える．

次に，術語を初歩的な用語に置き換えさえすれば簡単に証明できる定理の例を示そう．

(1) 集合 S は，ある整数 n について「S はちょうど n 個の要素を持っている」といえるとき，**有限**であるという．これを $\|S\| = n$ と書く——ここで $\|S\|$ は集合 S の要素の個数を表す．もし S が有限でなければ，S は**無限**であるという．直観的にいえば，無限集合とはどんな整数よりも多くの要素を含む集合のことである．

(2) S と T をある集合 U の部分集合とする．T が（U についての）S の**補集合**であるとは，次の条件が成り立つことをいう：$S \cup T = U$ でしかも $S \cap T = \varnothing$．いいかえれば，U のどの要素も，S と T のどちらか一方の中にある．これはまた,「T はちょうど，S の中にない要素から成る」ともいえる．

定理 1.5 S がある無限集合 U の有限部分集合であれば，S の補集合 T は無限集合である．

証明　直観的にいえば「あるものの無尽蔵の蓄え U から有限の量（S）を取り去っても，残りはまだ無限にある」ということである．まずこの定理の述べている事実を図 1.4 のように置き換えてみよう．

もとの表現	新しい表現
S は有限である．	ある整数 n について，$\|S\| = n$
U は無限である．	$\|U\| = p$ を満たす整数 p は存在しない．
T は S の補集合である．	$S \cup T = U$ でしかも $S \cap T = \varnothing$

図 1.4　定理 1.5 の言い換え

これだけでもまだ進めないので，よく使われる証明の技法「背理法」が必要である．その証明法では，1.3.3 項で詳しく述べるが，我々は「結論がまちがっている」と仮定する．そしてその仮定から，前提の一部を使い，与えられた前提文のどれかに反する文を証明する．これによって,「前提がすべて正しくしかも結論がまちがっている」ことはあり得ないことが示される．唯一の可能性は，前提が真の場合には結論も真だ，ということである．したがって，定理は正しい．

定理 1.5 の場合，結論を否定すると「T は有限である」になる．そこで T は有限であると仮定すると，前提（の一部）から S も有限で，ある整数 n について $\|S\| = n$ と

なる．同じように仮定「T は有限」を言い換えれば，ある整数 m について $\|T\| = m$ である．

さて，前提の中に $S \cup T = U$ でしかも $S \cap T = \varnothing$ が含まれている．だから U の要素は S と T のどちらか一方の中にある．$n+m$ は整数で $\|S \cup T\| = n+m$ であるから，U は有限である．詳しくいえば，U の要素はある整数に等しいが，それが「有限」の定義である．しかしこの「U は有限である」という文は，前提の中の「U は無限である」という文と矛盾する．このように，結論に矛盾する文から前提との矛盾を導けたので，「背理法」の原理によって，この定理は正しいと断言できる．▨

証明にこれほど言葉を費やすことはない．証明の背後にあるアイデアがわかれば，次のように短く書き直すことができる．

(定理 1.5 の) 証明　$S \cup T = U$ でしかも S と T は共通部分がないので，$\|S\| + \|T\| = \|U\|$ である．S は有限なので，ある整数 n について $\|S\| = n$ であり，U は無限であるから $\|U\| = p$ を満たす整数 p は存在しない．もし T が有限であると仮定すると，ある整数 m について $\|T\| = m$ となり，したがって $\|U\| = \|S\| + \|T\| = n+m$ となるが，これは「$\|U\| = p$ を満たす整数 p は存在しない」という前提に矛盾する．▨

1.2.3　定理の他の形

「もし…ならば」という形の定理は数学の一般的な分野で最もよく現れる．しかし他の形の文が定理として証明されることもある．この項では定理のよくある形と，それらの標準的な証明法を紹介する．

「もし…ならば」の言い換え　単純な「もし…ならば」型ではないが，実質的に同じこと「もし前提 H がある変数の値について正しければ，その同じ値について結論 C も正しい」を別の型で表現している定理もある．次に「もし H ならば C」を言い表すいくつかの方法を示す．

(1)　H は C を保証する（含意する）．

(2)　H であるのは C であるときに限る．

(3)　C は H にしたがう．

(4)　H であれば必ず，C も成り立つ．

(4) には「もし H が成り立つなら，C も正しい」とか「H が成り立つときはいつも，C も成り立つ」など，いろいろな変種がある．

例 1.6 定理 1.3 は，次のように表現できる．

(1) $x \geq 4$ は $2^x \geq x^2$ を保証する．

(2) $x \geq 4$ であるのは $2^x \geq x^2$ であるときに限る．

(3) $2^x \geq x^2$ は $x \geq 4$ にしたがう．

(4) $x \geq 4$ であれば必ず，$2^x \geq x^2$ も成り立つ． ▨

限定辞を含む文

多くの定理は「すべての」とか「ある…があって」のような限定辞を含んでいる（「すべての」を「任意の」とか「どんな…についても」と書くことも多い）．そのような場合，限定辞が文に現れる順序は文の意味に影響を及ぼす．二つ以上の限定辞を持つ文を理解するには,「すべての」担当と「ある」担当の 2 人で交互に関係する変数の値を決めるゲームを考えるとわかりやすい.「すべての」担当は，すべての可能な選択肢を考えなければならない．だからその選択は，ふつう変数のまま残される．しかし「ある」担当は，それ以前に選ばれた値に応じて，一つの値を選べばよい．限定辞が現れる順序は，2 人が選択を行う順序を決定する．最後に選択をしなければならない人がいつでもある適切な値を選べるなら，その文は正しい．

たとえば「無限集合」の別の定義を取り上げてみよう：集合 S が**無限**であるとは,「すべての正整数 n について，S のある部分集合 T があって，T はちょうど n 個の要素を持っている」ことである．ここでは「すべての」が「ある」より前なので，我々は任意の整数 n を考えなければならない．そして「ある」担当は，n の値を教わってから，適当な部分集合 T の選択に取りかかる．たとえば，S が整数の集合であれば,「ある」担当者は $T = \{1, 2, \cdots, n\}$ を選べば，n が何であろうと成功する．これで整数の集合が無限であることの証明ができた．

次の文は「無限」の別の定義のように見えるが，限定辞の順序が逆なので，不正確である：「S のある部分集合 T があって，すべての n について，T はちょうど n 個の要素を持っている.」今度は，S が整数の集合だとして,「ある」担当者がまず部分集合 T を何か選ぶ：たとえば $T = \{1, 2, 5\}$ が選ばれたとしよう．この選択に対して「すべての」担当者は，T が「すべての n に対して，n 個の要素を持つ」ことを示さなければならないが，そんなことは不可能である．たとえば $n = 4$ のときにこれは正しくないし，実は $n \neq 3$ であればいつも正しくない．

証明はどの程度きちんと書かねばならないか？

この問に答えるのはそうやさしくはない．証明の最低目標は「説得」であり，あなたの宿題の採点者であれあなた自身であれ，あなたが証明で使った作戦の正しさを納得させられればよい．説得力があれば，それで十分である．しかし証明の「消費者」を説得できなければ，省略が多すぎたのである．

証明の不確かさのある部分は消費者の予備知識の違いから来る．定理 1.4 の証明では，あなたに十分な知識があり，「もし $y \geq 1$ ならば $y^2 \geq 1$」のような文が理解できると仮定していた．あなたの知識が不十分だと，このような文もいくつかの段階を分けて演繹的に証明しなければならない．

しかし証明に必要なある事柄があり，それを省くと証明は確実に不適切になる．たとえば演繹的な証明の中で前提やすでに導いた文から正当化できない文を使うことは，適切ではあり得ない．必要十分条件文の証明では，十分性の証明を一つ，必要性の証明を一つ与えなければならない．もう一つ例を挙げれば，帰納的証明（1.4 節で扱う）では基礎部分の証明と帰納部分の証明を欠かすことができない．

なお形式論理学では「もし…ならば」をよく記号 $\rightarrow$ で表す．すなわち文「もし H ならば C」を「$H \rightarrow C$」と書くのである．しかし本書ではこの記号は使わない．

必要十分条件文　定理が「A であるための必要十分条件は，B である」という形をしていることもある．これは「A と B は（論理的に）同値である」,「A ならば B で，逆も成り立つ」と書かれることもある．これは二つの「もし…ならば」文，すなわち「もし A ならば B」と「もし B ならば A」とを合わせたものと考えてよい．だからこれを証明するには，次の二つの文を証明すればよい：

(1)　(十分性)：もし B ならば A,

(2)　(必要性)：もし A ならば B．これは「A であるのは B であるときに限る」といってもよい．

証明はどちらを先にしてもよい．多くの定理では一方が他方より決定的にやさしく，やさしい方を先に述べてそれをまず片づけるのがふつうである．

形式論理学では「必要十分」を記号 $\equiv$ あるいは $\Leftrightarrow$ で表す：すなわち $A \equiv B$ や $A \Leftrightarrow B$ は,「A であるための必要十分条件は B」を意味する．

必要十分条件文を証明するには,「十分」性と「必要」性の両方を証明しなければならない．ときには一つの同値性を，二つ以上の同値性に分解して証明し

たほうがラクな場合もある．すなわち，「A と B は同値」を証明するのに，まず「A と C は同値」を証明し，それから「C と B は同値」を証明するのである．それぞれの段階で必要性と十分性の両方を証明するのを忘れなければ，これでうまくいく．どちらかの段階で片方しか証明しないのだと，全体の証明が無効になる．

次に必要十分性の証明の例を示そう．ここで新しい記号を使う：

(1)　実数 x の**床** $\lfloor x \rfloor$ とは，x に等しいかまたは x より小さい最大の整数のことである．

(2)　実数 x の**天井** $\lceil x \rceil$ とは，x に等しいかまたは x より大きな最小の整数のことである．

定理 1.7　実数 x について，$\lfloor x \rfloor = \lceil x \rceil$ であるための必要十分条件は，x が整数であることである．

証明　(必要性) ここでは $\lfloor x \rfloor = \lceil x \rceil$ と仮定して，x が整数であることを証明したい．床と天井の定義から，$\lfloor x \rfloor \leq x$，$\lceil x \rceil \geq x$ である．仮定から $\lfloor x \rfloor = \lceil x \rceil$ なので，最初の不等式の床を天井に置き換えると $\lceil x \rceil \leq x$ となる．だから $\lceil x \rceil \leq x$ でしかも $\lceil x \rceil \geq x$ なので，数の不等式の性質から $\lceil x \rceil = x$ である．$\lceil x \rceil$ は整数であるから，x も整数でなければならない．

(十分性) 今度は x が整数であると仮定して，$\lfloor x \rfloor = \lceil x \rceil$ を証明したい．それは容易である．床と天井の定義から，$\lfloor x \rfloor$ も $\lceil x \rceil$ も x に等しく，したがって互いに等しい．▨

1.2.4　「もし…ならば」の形をしていない定理

ときには，前提のない定理に出会うことがある．三角法の有名な事実が，よい例である．

定理 1.8　$\sin^2 \theta + \cos^2 \theta = 1$　▨

実は，この定理には前提が**ある**ので，その前提はこの定理を解釈するのに必要なすべての事実の全体である．具体的にいうと，隠された前提は θ が角度で，関数サイン，コサインは角に対する通常の定義にしたがう，ということである．これらの関数の定義とピタゴラスの定理 (直角三角形の斜辺の2乗は，他の2辺の2乗の和に等しい) とから，この定理が証明できる．要するに，この定理を「もし…ならば」型できちんと述べれば，「もし θ が角度ならば，$\sin^2 \theta + \cos^2 \theta = 1$」となる．

1.3 証明のいろいろ

この節では，証明するための補足的な話題をいくつか取り上げる．

(1) 集合についての証明

(2) 背理法

(3) 反例による証明

1.3.1 集合の同一性の証明

オートマトン理論ではよく，二つの異なる方法で構成された集合が「同一である」といっている定理を証明する必要が起こる．多くの場合それらは文字列の集合（「言語」と呼ばれる）であるが，ここでは集合の性質は重要ではない．もし E, F が集合を表す二つの式であるなら，$E = F$ という主張は表現されている集合が同じであることを意味している．もっと詳しくいえば，E が表している集合の要素はすべて F が表している集合に属しているし，F が表している集合の要素はすべて E が表している集合に属している，ということである．

例 1.9 **和の交換法則**とは，二つの集合 R, S の和集合は順序によらないことである．式で書けば $S \cup R = R \cup S$ となる．この場合，E とは式 $S \cup R$, F とは式 $R \cup S$ のことである．和の交換法則は $E = F$ を主張する．▨

集合の等式 $E = F$ は，必要十分条件文で書き表すことができる：要素 x が E の中にある必要十分条件は，x が F の中にあることである．ここから二つの集合の一致 $E = F$ を主張する文の証明の方針が見えてくる；「必要十分条件文」の証明法にしたがえばよい：

(1) 「もし x が E の中にあるなら，x は F の中にもある」ことを証明する．

(2) 「もし x が F の中にあるなら，x は E の中にもある」ことを証明する．

この証明過程の例として，**和から積への分配法則**を証明してみよう．

定理 1.10 $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$

証明 関係する二つの集合表現は，

$$E = R \cup (S \cap T) \text{ と } F = (R \cup S) \cap (R \cup T)$$

である．まず x が E の中にあると仮定して，x が F の中にもあることを証明しよう．この過程は図 1.5 に示されているが，和と積（共通部分）の定義はわかっているものとする．

次に，その逆を証明しなければならない．ここでは x が F の中にあると仮定して，

	文	根拠
(1)	x は $R \cup (S \cap T)$ の中にある．	前提
(2)	x は R かまたは $S \cap T$ の中にある．	(1) と和の定義
(3)	x は R の中にあるか，または x は S と T の両方の中にある．	(2) と積の定義
(4)	x は $R \cup S$ の中にある．	(3) と和の定義
(5)	x は $R \cup T$ の中にある．	(3) と和の定義
(6)	x は $(R \cup S) \cap (R \cup T)$ の中にある．	(4), (5) と積の定義

図 1.5　定理 1.10 の証明の前半

	文	根拠
(1)	x は $(R \cup S) \cap (R \cup T)$ の中にある．	前提
(2)	x は $R \cup S$ の中にある．	(1) と積の性質
(3)	x は $R \cup T$ の中にある．	(1) と積の性質
(4)	x は R の中にあるかまたは， x は S，T の両方の中にある．	(2), (3) と和についての推論
(5)	x は R かまたは $S \cap T$ の中にある．	(4) と積の性質
(6)	x は $R \cup (S \cap T)$ の中にある．	(5) と和の性質

図 1.6　定理 1.10 の証明の後半

E の中にもあることを証明する．その要領は図 1.6 に示されている．必要十分性の両方が証明できたので，上の分配法則が証明できた．▨

1.3.2　対　　偶

「もし…ならば」型の文には，それと同値な，場合によってはそれより証明しやすい言い換えがある．文「もし H ならば C」の対偶とは，「もし C でなければ H でない」という文のことである．どんな文でもその対偶と，いつでも真・偽が一致するので，一方を証明すれば他方も証明できたことになる．

「もし H ならば C」と「もし C でなければ H でない」とが論理的に同値であることは，次の四つの場合を観察してみればわかる．

(1)　H も C もともに正しい．

(2)　H は正しいが C は誤りである．

(3)　C は正しいが H は誤りである．

(4)　H も C も誤りである．

「もし H ならば C」が偽になるのは，一つの場合しかない：(2) のように前提が正しく，結論が誤っている場合である．それ以外の三つの場合は，結論が誤っている (4) も含めて，この条件文は正しい．

では対偶「もし C でなければ H でない」はいつ偽になるのだろうか．これが偽になるのは，前提（C でない）が正しく，結論（H でない）が偽である場合だけである．しかし「C でない」が正しいとはまさに C が偽であるときで，「H でない」が偽であるとは H が真である場合に他ならない．それは再び (2) の場合であるから，もとの文と対偶とは，四つある場合のすべてについて「ともに真」か「ともに偽」である．すなわち，これらは論理的に同値である．

例 1.11 定理 1.3 を思い出してみよう：「もし $x \geq 4$ ならば $2^x \geq x^2$」この文の対偶は「もし $2^x \geq x^2$ でなければ $x \geq 4$ でない」である．もっとこなれた表現では「$a \geq b$ でない」は「$a < b$」と同じであるから，対偶は「$2^x < x^2$ ならば $x < 4$」となる．▨

集合についての「必要十分」の言い換え

すでに述べたように，集合を表す式の一致を主張する定理は，必要十分条件文と考えてよい．だから定理 1.10 は次のように述べられる：要素 x が $R \cup (S \cap T)$ の中にある必要十分条件は，x が

$$(R \cup S) \cap (R \cup T)$$

の中にあることである．

集合の一致を表すもう一つのよく使われる（英語の）表現は，慣用句「すべて，またそれ以外にない」（all-and-only）である．たとえば定理 1.10 は，次のようにも述べられる：$R \cup (S \cap T)$ の要素はすべて $(R \cup S) \cap (R \cup T)$ の要素であり，またそれ以外にはない．

逆

用語「対偶」を「逆」と混同してはいけない．条件文の**逆**とは，ただ順序を入れ換えただけである：すなわち，「もし H ならば C」の逆は「もし C ならば H」である．対偶はもとの文と論理的に同値であるが，逆はもとの文と同値とは限らない．必要十分条件文の証明において，必要性と十分性はいつも互いに逆の関係にある．

必要十分条件文を証明したいとき，対偶を利用するといろいろな選択肢ができる．たとえば，集合の同一性 $E = F$ を証明したいとき，「もし x が E の中にあれば F の中にもあるし，x が F の中にあれば E の中にもある」を証明する代わりに，どちらかをその対偶で置き換えてもよい．一つの等価な形は：

- もし x が E の中にあれば F の中にもあり，E の中になければ F の中にもない．

1.3.3 背理法

「もし H ならば C」型の文を証明するもう一つの方法は，

- H でありしかも C でないとすると，矛盾が生じる

のを証明することである．すなわちまず前提 H と，結論 C の否定の両方を仮定するのである．これらから，正しくないとわかっている何かが導ければ，証明が完成する．これを**背理法**という．

例 1.12　定理 1.15 では，$H =$「U は無限集合，S は U の有限部分集合，T は U についての S の補集合」という前提の条件文を証明した．結論 C は「T は無限である」であった．その証明は背理法で進められた．我々は「C でない」，つまり「T は有限である」と仮定した．

あのときの証明は，H と C の否定から矛盾を導いた．我々はまず S も T も有限であると仮定すると，U も有限になることを証明した．しかし前提 H の中で U は無限であると述べられているので，集合が同時に無限であり有限でもあることはあり得ない以上，論理的に正しくない文が証明されたわけである．論理学の用語によれば，命題 P（U は有限である）と P の否定「P でない」（U は無限である）が両方導かれた．そして「P でありしかも P でない」は論理的に「偽」と同値であることを利用した．▨

背理法がなぜ論理的に正しいかを理解するために，1.3.2 項で考えた前提 H と結論 C の真理値の四つの組み合わせを思い出してほしい．H が真で C が偽である (2) の場合だけ，条件文「もし H ならば C」は偽になるのであった．「H でありCでない」と仮定して矛盾を導くことによって，(2) の場合はあり得ないことが示される．だから H と C の真理値の可能な組み合わせは，「もし H ならば C」が真となる三つの場合しかない．

1.3.4 反　例

実生活では，定理の証明を頼まれることなどない．それよりは，何かもっともらしいことを見せられて——たとえばプログラムを作る方針など——その「定理」が正しいかどうかを決めなければならないことが多い．この問題を解くには，定理の証明を試みたり，うまくいかなければそれが正しくないことを証明しようとするなど，いろいろやってみないといけない．

定理はふつう，その変数のすべての値など，無数に多くの場合についての主張である．実際，厳密な数学の慣習では「定理」の名に値するのは無限の場合を含む文だけである；変数を含まない文や，その変数の有限個の値にしかあてはまらない文は**観察**（observation）と呼ばれる．ある定理と称する文が定理でないことを示すためには，それがたった一つの場合でも正しくないことを示せばよい．それはプログラムがある一つの入力に対して期待される動作をしなければ，虫がいると考えられるのと同じことである．

ある文が定理でないことを示すのは，定理であることを証明するよりやさしいことが多い．次の二つの例では，最初のものは明らかに定理ではないが，次の文は惜しいところで定理になり損ねたので，定理かどうかの判定には少し考えなければならない．

怪しい定理 1.13　すべての素数は奇数である（もっと形式的にいうと：「もし整数 x が素数なら，x は奇数である」）．

反証　整数 2 は素数であるが，2 は偶数である．▨

次に，剰余計算にかかわる「定理」を考える．それにはまず基本的な定義を述べなければならない．a と b が正整数であるとき，$a \bmod b$ とは a を b で割った余り r，すなわちある整数 q について $a = qb + r$ を満たす 0 以上 b 未満の整数 r のことである．たとえば $8 \bmod 3 = 2$，$9 \bmod 3 = 0$ である．我々の最初の定理は，あとで誤りであることを示すが，次の通りである．

怪しい定理 1.14　$a \bmod b = b \bmod a$ であるような整数 a，b の対は存在しない．▨

ここでの a と b のように，何かの対にかかわることを調べるには，二つの関係を対称性によって簡単にできることが多い．この場合，我々は $a<b$ の場合にしぼって検討をすればよいので，$b<a$ の場合は a と b を入れ換えれば，(怪しい）定理 1.14 と同じ等式が得られる．しかし $a=b$ という，第3の場合にも注意しなければならない．この場合こそ，我々の証明の努力に致命的である．

まず $a<b$ と仮定しよう．すると $a=0\times b+a$，つまり $q=0$，$r=a$ なので，$a \bmod b$ の定義から $a \bmod b=a$ である．一方 $x \bmod a$ は，x が何であろうと a よりは小さいので，$a \bmod b<a$ である．このように $a<b$ の場合は $b \bmod a<a \bmod b$ なので，$a \bmod b=b \bmod a$ にはなり得ない．

これと対称な議論から，もし $b<a$ ならば $a \bmod b \neq b \bmod a$ であることもわかる．

しかし，第3の場合 $a=b$ を考えると，任意の x について $x \bmod x=0$ であるから，もし $a=b$ であればたしかに $a \bmod b=b \bmod a$ となる．これでこの「怪しい定理」の反証ができた．

（怪しい定理 1.14 の）反証　$a=b=2$ とすると

$$a \bmod b=b \bmod a=0 \quad ▨$$

反例を探しているうちに，我々はこの怪しい定理が成り立つ正しい条件を見つけた．次に正しい形の定理とその証明を述べる．

定理 1.15　$a \bmod b=b \bmod a$ であるための必要十分条件は，$a=b$ である．

証明　（十分性）$a=b$ と仮定する．上で観察したように，任意の整数 x について $x \bmod x=0$ である．だから $a=b$ であればつねに $a \bmod b=b \bmod a$

（必要性）今度は $a \bmod b=b \bmod a$ と仮定しよう．ここでは背理法が適しているので，結論の否定 $a\neq b$ も仮定する．すると $a=b$ の可能性はなくなるので，$a<b$ の場合と $b<a$ の場合とを考えればよい．

我々はすでに，もし $a<b$ ならば $a \bmod b=a$ でしかも $b \bmod a<a$ であり，したがって $a \bmod b=b \bmod a$ ではあり得ないことを見た．だから前提と結論の否定から，矛盾が導かれる．

対称性によって，もし $b<a$ ならば $b \bmod a=b$ でしかも $a \bmod b<b$ である．ここからも矛盾が導かれるので，必要性も正しいことがわかる．両方向の証明ができたので，定理の証明を終わる．▨

1.4 帰納的証明

特殊な形の証明で,「帰納的」と呼ばれるものがある．これは再帰的に定義された対象を扱うときに欠くことができない．よく知られた帰納的証明の多くは自然数を扱っているが，オートマトン理論でも木とか各種の式（たとえば 1.1.2 項で簡単に触れた正則表現）のように再帰的に定義される概念についての証明には帰納的証明が必要になる．この節では，帰納的証明の手法をまず整数についての「単純な」帰納法から導入しよう．それから再帰的に定義された諸概念について,「構造的」帰納法をどのように行うかを示す．

1.4.1 整数についての帰納法

整数 n についてのある条件 $S(n)$ を証明したい，としよう．よくある手順にしたがえば，次の二つのことを証明する．

(1) **基礎**：ある整数 i について，$S(i)$ が正しいことを示す．ふつうは $i=0$ または $i=1$ であるが，i が小さいところではいくつか S が成り立たない場合もあるために，もっと大きな i から始めることもある．

(2) **帰納的ステップ**：$n \geq i$ と仮定して,「もし $S(n)$ ならば $S(n+1)$」であることを証明する．ここで i は基礎となる整数である．

これらの二つから，基礎となる整数 i 以上のすべての整数 n について $S(n)$ が正しいことは，直観的には明らかであろう．それは次のように論じることもできる．$S(n)$ が i 以上のいくつかの整数について正しくないとする．そのような整数 n の中で最小のものを j とする：$S(j)$ は誤りで，$j \geq i$ である．この j は，i ではあり得ない——基礎の段階で，$S(i)$ が正しいことは証明済みだからである．だから j は i よりも大きい．すると $j-1 \geq i$ であるから，$S(j-1)$ は正しい．

ところで帰納的ステップで，もし $n \geq i$ ならば，$S(n)$ から $S(n+1)$ が導かれることを証明した．だから $n=j-1$ とおけば，$S(j-1)$ から $S(j)$ が導かれる．$S(j-1)$ は正しいので，$S(j)$ も正しい．

我々は証明したいことの否定を仮定した；$S(j)$ がある $j \geq i$ について正しくない，と仮定したのである．しかし j が何であっても矛盾が導かれたので，背理法の原理によって，$S(n)$ はすべての $n \geq i$ について正しい．

残念ながら，この議論には微妙な論理的欠陥が隠されている．「$S(j)$ が成り立たないような最小の j がある」という仮定は，まず第一に帰納法の根本的な原理を信じるから成り立つことである．別のいい方をすると，そのような j が存在することを証明する唯一の方法は，本質的に帰納的証明によるしかない．しかし，上に述べた「証明」は直観的に十分明確で，現実世界の理解とも適合する．そこで一般に次のことが，論理的推論体系の必須の一部として認められている．

- **帰納法の原理**：もし $S(i)$ が証明でき，しかもすべての $n \geq i$ について「$S(n)$ ならば $S(n+1)$」であることも証明できたならば，$S(n)$ はすべての $n \geq i$ について正しいと結論してよい．

次に整数についての定理を帰納法で証明する二つの例を挙げる．

定理 1.16　すべての $n \geq 0$ について，

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

証明　証明は二つの部分に分かれる：基礎と帰納である；それらを順に証明しよう．
基礎：基礎としては，$n=0$ を選ぶ．$n=0$ としても定理が意味を持つことに驚く人もいるかも知れない：左辺が「$i=1$ から 0 までの和」になってしまうからである．しかし和の上限（終点，ここでは 0）が下限（出発点，ここでは 1）より小さい場合には，総和 $\sum$ は 0 個の項の和と考えて，0 とみなす習慣がある．それにしたがえば左辺 $\sum_{i=1}^{0}$ は 0 である．

一方，式 (1.1) の右辺の値は $0 \times (0+1) \times (2 \times 0 + 1) = 0$ である．したがって，等式 (1.1) は $n=0$ の場合には成り立つ．
帰納：今度は $n \geq 0$ と仮定する．目標は帰納法の段階，すなわち「等式 (1.1) が正しければ，その中の n を $n+1$ に置き換えた式も正しい」ことである．後者は

$$\sum_{i=1}^{n+1} i^2 = \frac{(n+1)\{(n+1)+1\}\{2(n+1)+1\}}{6} \tag{1.2}$$

である．式 (1.1)，(1.2) の右辺を展開して整理すると，次のようになる：

$$\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6} \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = \frac{2n^3+9n^2+13n+6}{6} \tag{1.4}$$

我々の目標は，式 (1.3) を使って式 (1.4) を証明することである——これらがそれぞれ，帰納的証明の $S(n)$ と $S(n+1)$ にあたるものである.「鍵」は，$n+1$ までの和を表す式 (1.4) の左辺を，n までの和と第 $n+1$ 項の和に分解することである．そうすれば，n までの和は式 (1.3) の右辺に置き換えられ，式 (1.4) が正しいことがわかる．それを式で書けば，次のようになる．

$$\sum_{i=1}^{n} i^2 + (n+1)^2 = \frac{2n^3+9n^2+13n+6}{6} \tag{1.5}$$

$$\frac{2n^3+3n^2+n}{6} + (n^2+2n+1) = \frac{2n^3+9n^2+13n+6}{6} \tag{1.6}$$

最後の等式 (1.6) の確認は，左辺の多項式の計算を行って右辺に等しいことを示しさえすればよい．▨

例 1.17 次の例として，1.2.1 項で扱った定理 1.3 を証明する．これは「もし $x \geq 4$ ならば $2^x \geq x^2$」という定理であった．前の直観的な証明では，x が 4 を越えて大きくなるにつれて比 $x^2/2^x$ がどんどん小さくなることを利用した．同じことが，基礎 $x=4$ から出発して $2^x \geq x^2$ であることを示す帰納的証明ではより正確になる．なおこの不等式は，$x<4$ では成り立たないことを注意しておこう．

基礎：もし $x=4$ ならば，2^x も x^2 も，どちらも 16 であり，$2^x \geq x^2$ が成り立つ．

帰納：ある $x \geq 4$ について $2^x \geq x^2$ であると仮定しよう．これを仮定して証明したいのは，同じ式の x を $x+1$ に置き換えたもの，つまり $2^{x+1} \geq (x+1)^2$ である．これらが帰納的証明の $S(n)$ と $S(n+1)$ に相当する；変数として n でなく x を使っているのに気を取られてはいけない；x や n は単なる局所的変数である <**訳注**：名前はどうでもいい>.

定理 1.16 の証明と同じように，$S(x+1)$ を書き換えて，前提 $S(x)$ が使いやすいようにしよう．今度は，$2^{[x+1]}$ を 2×2^x と書くことができる．$S(x)$ から $2^x \geq x^2$ がいえるので，$2^{x+1} \geq 2 \times 2^x \geq 2x^2$ が導かれる．

しかしこれは欲しいものとは違う：証明したいのは $2^{x+1} \geq (x+1)^2$ なのである．これを導く一つの方法は，$2x^2 \geq (x+1)^2$ を証明し，$\geq$ の性質（推移性）から $2^{x+1} \geq 2x^2 \geq (x+1)^2$ を示すことである．その

$$2x^2 \geq (x+1)^2 \tag{1.7}$$

再帰的に定義された概念としての整数

帰納的な証明は，対象が再帰的に定義されている場合に有効である，と述べた．しかし最初の例は整数についての帰納法であったが，整数はふつう「再帰的に定義される」とは思われていない．ただ非負整数に限れば，自然な再帰的定義があって，「先に定義された対象からあとで定義される対象へと進む」形が整数の帰納法の進行とよく合っている．

基礎：0 は整数である．

再帰：もし n が整数ならば，$n+1$ も整数である．

の証明には，仮定 $x \geq 4$ を使ってよい．式 (1.7) を整理すると

$$x^2 \geq 2x + 1 \tag{1.8}$$

になるが，この両辺を x で割ると

$$x \geq 2 + \frac{1}{x} \tag{1.9}$$

となる．$x \geq 4$ であるから，$1/x$ は $1/4$ より小さい．式 (1.9) の左辺は少なくとも 4 であり，右辺はせいぜい 2.25 である．これで式 (1.9) の正しさが証明できた．ここから式 (1.8)，(1.7) も正しいことがわかる．そして式 (1.7) から $2x^2 \geq (x+1)^2$ が導かれ，最初の結果とあわせて $2^{x+1} \geq (x+1)^2$ ，すなわち $S(x+1)$ が得られる．▨

1.4.2　整数についての帰納法の，より一般的な形

帰納法の証明は，ときには 1.4.1 項で説明した方式——$S(n)$ をある基礎になる値について証明し，それから「$S(n)$ ならば $S(n+1)$」を示す——ではうまくいかないことがある．そこでこれまでの方式の，二つの重要な一般化を示そう：

(1)　複数個の基礎を使うこともできる．すなわち，まずある $j > i$ について $S(i), S(i+1), \cdots, S(j)$ を証明する．

(2)　$S(n+1)$ の証明において，$S(n)$ だけでなく，

$$S(i),\ S(i+1),\ \cdots,\ S(n)$$

がすべて正しいことを使うことができる．さらに，もし基礎が $S(j)$ に及んでいるなら，$n \geq i$ でなくて $n \geq j$ を仮定してよい．

これらの基礎と帰納から，すべての $n \geq i$ について $S(n)$ が成り立つことが導かれる．

例 1.18 これらの拡張の双方の威力を知るために，次の例を取り上げる．証明したい文 $S(n)$ は,「$n \geq 8$ ならば，n は 3 と 5 の和で表せる」というものである．なおついでながら，7 は 3 と 5 の和では表せない．

基礎：基礎となるのは $S(8)$，$S(9)$ と $S(10)$ である．証明は順に $8 = 3+5$，$9 = 3+3+3$，$10 = 5+5$ でよい．

帰納：$n \geq 10$ と仮定し，さらに $S(8), S(9), \cdots, S(n)$ が正しいとする．これらの前提から，$S(n+1)$ を導かなければならない．我々の作戦は $n+1$ から 3 を引くことで，その数が 3 と 5 の和で表せることから，もう一つ 3 を加えれば $n+1$ が表せることが導ける．

もっと形式的にいえば，$n-2 \geq 8$ であるから，$S(n-2)$ を仮定してよいことに注意しよう．すなわち，$n-2 = 3a + 5b$ を満たす非負整数 a，b がある．だから $n+1 = 3+3a+5b$ なので，$n+1$ は $a+1$ 個の 3 と b 個の 5 の和で表せる．これで $S(n+1)$ が導かれ，帰納のステップを終わる．▨

1.4.3 構造的帰納法

オートマトン理論では，再帰的に定義されたいくつかの構造について定理を証明したいことがある．おなじみの木とか式などが，重要な例である．帰納法のように，再帰的な定義は基礎的な場合と再帰の段階に分かれ，基礎ではいくつかの基本的な構造が定義され，再帰の段階では前に定義された構造からより複雑な構造が定義される．

例 1.19 まず木の定義を示す．

基礎：ただ一つの節点は木である．その節点はその木の**根**と呼ばれる．

再帰：k をある正整数とする．もし $T_1, T_2, \cdots, T_k$ が木ならば，次のようにして新しい木を構成できる：

(1) 新しい節点 N から始める．これが新しい木の根である．

(2) すべての木 $T_1, T_2, \cdots, T_k$ のコピーをつけ加える．

(3) N から木 $T_1, T_2, \cdots, T_k$ のそれぞれの根への辺をつけ加える．

図 1.7 はより小さい木から根 N をもつ木への構成を示している．

例 1.20 再帰的な定義のもう一つの例を挙げよう．今度は算術演算 $+$，$*$ を使う式を定義する．

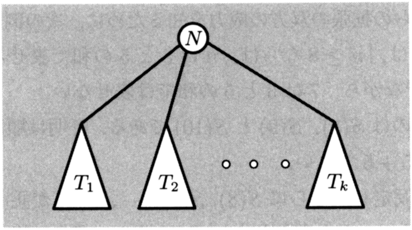


図 1.7　木の再帰的構成

基礎：数あるいは文字（すなわち変数）はすべて式である．

再帰：もし E と F が式であれば，$E+F$，$E*F$，(E) も式である．

たとえば基礎によって，2 も x も式である．再帰のステップから，$x+2$，$(x+2)$ またさらに $2*(x+2)$ もみな式である．これらの式が前のものにどのように依存しているのか注意すること．▨

再帰的な定義に対して，次のような形の帰納的証明が適用できる．これは**構造的帰納法**と呼ばれている．$S(X)$ をある構造 X についての文とし，構造 X は次のような形で再帰的に定義されている，とする．

(1) **基礎**：基礎的な構造 X（複数個あるかも知れない）について，$S(X)$ を証明する．

(2) **帰納**：構造 X が，たとえば $Y_1, Y_2, \cdots, Y_k$ から構成されるときは，$S(Y_1), S(Y_2), \cdots, S(Y_k)$ を仮定して，$S(X)$ を証明する．＜**訳注**：これ

構造的帰納法の背後にある直観

構造的帰納法がなぜ正しいのか，直観的な説明はできる．再帰的な定義によって，ある構造 $X_1, X_2, \cdots$ が定義を満たすことが，一つずつ確かめられたとする．基礎となる要素が最初に現れ，そのあとの X_i が定義を満たす（定義された構造の集合に属している）ことは，このリストで X_i より前にある，定義を満たす要素に依存する．そのように考えると，構造的帰納法とは整数についての文 $S(X_n)$ についての帰納法と何ら変わらない．この帰納法は 1.4.2 項で議論した一般化された形でよいので，基礎が複数個あり，帰納ではそれ以前のすべての場合を利用できる．しかし 1.4.1 項でも注意したように，この直観は形式的な証明にはならない．実際，この直観の妥当性は，前の項でもとの帰納法の妥当性を仮定したように，やはり仮定しなければならない．

をすべての構成法について行う．>

これによって，「すべての X について $S(X)$ が正しい」と結論してよい．次に木と式について成り立つ証明できる事実の例として，二つの定理を示す．

定理 1.21 どの木でも，辺より節点のほうが一つだけ多い．

証明 我々が構造的帰納法で証明したい $S(T)$ とは：「もし T が木で，T が n 個の節点と e 個の辺を持っているなら，$n = e + 1$」

基礎：基礎となるのは，T がただ一つの節点から成る場合である．この場合 $n = 1$，$e = 0$ であるから，関係 $n = e + 1$ はたしかに成り立つ．

帰納：T が再帰的ステップによって，より小さい木 $T_1, T_2, \cdots, T_k$ から構成されたとする．我々は $S(T_i)$ が $i = 1, 2, \cdots, k$ について正しいと仮定する：T_i が n_i 個の節点と e_i 個の辺を持っているとすると，つねに $n_i = e_i + 1$ である，ということである．

T の節点は N と T_i のすべての節点を合わせた全体で，合計 $1 + n_1 + n_2 + \cdots + n_k$ 個になる．一方，T の辺は，再帰であとからつけ加えた k 個の辺と，T_i の辺を合わせた全体である．だから，T は

$$k + e_1 + e_2 + \cdots + e_k \tag{1.10}$$

個の辺を持つ．T の節点を表す式の中の n_i に $e_i + 1$ を代入すると，

$$1 + (e_1 + 1) + (e_2 + 1) + \cdots + (e_k + 1) \tag{1.11}$$

となる．そこで式 (1.11) の k 個の "+1" をまとめると，次のようになる：

$$k + 1 + e_1 + e_2 + \cdots + e_k \tag{1.12}$$

これは T の辺の個数を表す (1.10) より 1 だけ多い．したがって，T は辺より一つだけ多い節点を持つ．▨

定理 1.22 どんな式でも，左右の括弧の個数は一致する．

証明 形式的には，例 1.20 で定義される式 G について証明したい文 $S(G)$ は：G の中の左括弧の数と右括弧の数は一致する．

基礎：G が基礎で定義される式ならば，G は数か文字である．どちらも 0 個の左括弧と 0 個の右括弧を含むので，数はたしかに一致する．

帰納：定義の再帰的ステップにしたがえば，式 G を構成する規則は三つある．

(1) $G = E + F$

(2) $G = E * F$

(3) $G = (E)$

ここで $S(E)$ と $S(F)$ が正しいことは仮定してよい：つまり E には同じ数，たとえば n 個ずつの左右の括弧があり，F には m 個ずつの左右の括弧がある，としてよい．す

るとそれぞれの場合について，G の左右の括弧の数が計算できる：

(1) もし $G = E + F$ ならば，G は $n + m$ 個の左括弧と $n + m$ 個の右括弧を含む；n 個は E の，m 個は F の中にあったものである．

(2) もし $G = E * F$ ならば，G の左右の括弧の数はやはり $n + m$ 個である：理由は上と同じ．

(3) もし $G = (E)$ ならば，G は $n + 1$ 個の左括弧を持つ——一つは明示されているもので，n 個は E の中にある．同様に，G は $n + 1$ 個の右括弧を持つ；一つは明示され，n 個は E の中にある．

どの場合にも，G の左右の括弧は同数であることがわかった．この観察によって，証明が完成する．▨

1.4.4 相互的帰納法

ある文を単独では証明できず，いくつかの文 $S_1(n), S_2(n), \cdots, S_k(n)$ をまとめて n についての帰納法で扱った方がよい場合もある．オートマトン理論ではそのような例がたくさん見られる．例 1.23 では，あるオートマトンの働きを説明するのに，状況ごとに設定された一群の文を証明しなければならないという，よくある状況の例を示す．それらの文は，どのような入力列によってオートマトンがある特定の状態に到達するかを述べている．

厳密にいえば，一群の文を証明するとは，それらの論理積（AND 結合）を証明することである．たとえば一群の文 $S_1(n), S_2(n), \cdots, S_k(n)$ は，一つの文「$S_1(n)$ でしかも $S_2(n)$ でしかも $\cdots$ でしかも $S_k(n)$」に置き換えられる．しかし，証明すべき文がまったく独立なときは，基礎と帰納の各段階で，それらをまとめずに個別に扱った方が，混乱を軽減できることが多い．この種の証明は相互的帰納法と呼ばれる．次の例で，**相互的帰納法**で必要なステップを説明する．

例 1.23 例 1.1 のオートマトンで表される，オン／オフ・スイッチをまた取り上げてみよう．このオートマトンは図 1.8 で表現できる．ボタンを押すことで状態はオンとオフが切り替わり，最初の状態はオフである．すると次の文の組み合わせで，このスイッチの動作が説明できる，と思われる：

$S_1(n)$：n 回の入力（ボタンを押すこと）でこのオートマトンが状態オフにある必要十分条件は，n が偶数であることである．

$S_2(n)$：n 回の入力でこのオートマトンが状態オンにある必要十分条件は，n が奇数であることである．

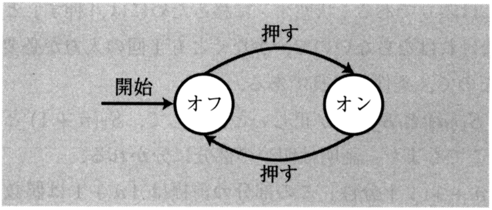


図 1.8　図 1.1 のオートマトン（再掲）

整数 n が同時に偶数かつ奇数にはならないので，S_1 ならば S_2 であり，逆も成り立つと考えてよさそうである．しかしオートマトンでは，それが一つの状態にあり，しかもただ一つの状態に限るとは必ずしもいえない．図 1.8 のオートマトンはいつもある特定の状態にあるが，その事実は相互的帰納法の一部で証明されなければならない．

S_1 と S_2 を証明するための基礎と帰納の各部分は，以下に示す通りである．証明には，偶数と奇数についてのいくつかの事実を利用する：ある偶数（奇数）に 1 を加えると奇数（偶数）になり，偶数（奇数）から 1 を引くと奇数（偶数）になる．

基礎：基礎として，$n=0$ を選ぶ．二つの文は，それぞれ両方向で証明しなければならない（S_1 も S_2 も必要十分条件文である）ので，基礎は四つの部分に分かれる（帰納も同様である）．

(1)　［$S_1(0)$；十分性］0 は事実偶数であるから，0 回の入力で図 1.8 のオートマトンが状態オフにあることを示さなければならない．オフは初期状態であるから，このオートマトンは 0 回の入力の後，たしかに状態オフにある．

(2)　［$S_1(0)$；必要性］オートマトンは 0 回の入力で，たしかに状態オフにあるので，0 が偶数であることを示さなければならない．しかし 0 が偶数であることは偶数の定義から明らかで，特に証明すべきことは何もない．

(3)　［$S_2(0)$；十分性］S_2 の十分性の前提は，「0 は奇数である」である．前提 H が誤りなら，1.3.2 項で論じたように，「もし H なら C」の形の文はすべて真である．だから基礎のこの部分は正しい．

(4)　［$S_2(0)$；必要性］前提「0 回の入力で，このオートマトンは状態オン

にある」は誤りである：状態オンに移るためには,「押す」と書かれた矢印を進まなければならないので，少なくとも 1 回の入力が必要である．前提が誤りなので，条件文は真である．

帰納：今度は $S_1(n)$ と $S_2(n)$ が正しいと仮定して，$S_1(n+1)$ と $S_2(n+1)$ を証明する．ここでもまた，証明は四つの部分に分かれる．

(1)　[$S_1(n+1)$；十分性] この部分の前提は「$n+1$ は偶数」である．したがって，n は奇数である．$S_2(n)$ によれば，n 回の入力のあとにオートマトンは状態オンにある．オンからオフへの矢印は，$n+1$ 回目の入力でオートマトンの状態がオフになることを示している．これで $S_1(n+1)$ の十分性が証明できた．

(2)　[$S_1(n+1)$；必要性] 前提は「$n+1$ 回の入力でオートマトンはオフの状態にある」である．図 1.8 をよく見ると，状態オフになるためにはその前の状態はオンであり，そこで入力を受けるしかないことがわかる．だから $n+1$ 回目の入力で状態オフにあるとしたら，n 回目の入力の後では状態はオンである．一方 $S_2(n)$ によれば，n は奇数でなければならない．したがって $n+1$ は偶数で，これこそ望みの「$S_1(n+1)$ の必要性の結論」である．

(3)　[$S_2(n+1)$；十分性] この部分は S_1 と S_2 の役割が逆になり，偶数と奇数が入れ替わるだけで，本質的に（1）と変わりない．読者は容易にこの部分を構成できるであろう．

(4)　[$S_2(n+1)$；必要性] この部分は S_1 と S_2 の役割が逆になり，偶数と奇数が入れ替わるだけで，本質的に（2）と変わりない．▨

例 1.23 から，相互的帰納法のパターンが抽出できる．

- それぞれの文は基礎・帰納の各段階で，独立に証明しなければならない．
- 必要十分条件文は，基礎・帰納の各段階で両方向とも証明しなければならない．

1.5 オートマトン理論の中心概念

この節では，オートマトン理論のいたるところで使われる最も重要な用語の定義を述べる．それらの概念には「アルファベット」（記号の集合）,「文字列」

（一つのアルファベットから選ばれた記号の列），それから「言語」（同じアルファベット上の文字列の集合）などがある．

1.5.1 アルファベット

アルファベットとは，記号の空でない有限集合のことである．アルファベットはふつう，記号 Σ で表される．よく使われるアルファベットには，次のものがある．

(1) $\Sigma = \{0, 1\}$，2 進アルファベット．

(2) $\Sigma = \{a, b, \cdots, z\}$，すべての小文字の集合．

(3) すべてのアスキー記号の集合，あるいはすべての印字可能なアスキー記号の集合．

1.5.2 文　字　列

文字列（あるいは**語**）とは，あるアルファベットから選ばれた記号の有限列のことである．たとえば，01101 は 2 進アルファベットからの文字列である．111 も，同じアルファベットからのもう一つの文字列である．

空列　　**空列**とは 0 個の記号から成る文字列のことである．空列は，記号 ε で表されるが，どんなアルファベットから選ばれたとしてもよい．

列の長さ　　文字列をその**長さ**，すなわちその文字列中の「記号の位置の数」で分類すると便利なことがある．たとえば列 01101 の長さは 5 である．列の長さとは，その中の「記号の個数」ともいわれるが，これは厳密とはいえない．列 01101 の中には二つの記号 0, 1 しかないが，記号の位置は五つあるので，長さは 5 である．しかし一般的には,「記号の数」は「記号の位置の数」と解釈できる，と考えてよい．

文字列 ω の長さを表す標準的な記法は $|\omega|$ である．たとえば $|011| = 3$, $|\varepsilon| = 0$ である．

アルファベットのベキ　　アルファベット Σ に対して，そのアルファベットから作れる特定の長さの記号列の集合を指数記法で表すことがある：Σ^k は，Σ から作れる長さ k の列全体の集合を表す．

例 1.24 Σ が何であっても，$\Sigma^0 = \{\varepsilon\}$ である．つまり，長さ 0 の列は ε だけである．

もし $\Sigma = \{0, 1\}$ ならば，$\Sigma^1 = \{0, 1\}$，$\Sigma^2 = \{00, 01, 10, 11\}$，

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\},$$

等々，である．Σ と Σ^1 とはいくらか混乱する恐れがある．前者はアルファベットで，その要素は記号 0, 1 である．後者は記号列の集合で，その要素は長さ 1 の記号列 0, 1 である．ここではこれら二つの集合を区別する記法は考えないが，前後関係から $\{0, 1\}$ のような集合がアルファベットなのか記号列の集合なのかはわかると思う．▨

＜**訳注：**Σ と Σ^1 を区別するなら記号の 0, 1 と記号列の 0, 1 とを区別しなければならない．ソフトウェアによっては「記号列の末尾に特別の記号 Null をつける」ものがあり，空列は 1 個の “Null”，長さ 1 の記号列 1 は “1 Null” で表されるので，明確に区別される．＞

アルファベット Σ 上のすべての列の集合は，ふつう Σ^* で表される．たとえば

$$\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \cdots\}$$

これは次のように表せる：

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

記号列の集合から，空列 ε を外したいときもある．空でない列全体の集合は Σ^+ で表される．これに関連する二つの等式は：

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$
- $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

列の連接　x と y をある列とする．x のコピーのあとに y のコピーをつないで得られる列を x と y の**連接**といい，xy で表す．詳しくいうと，x が i 個の記号の列 $x = a_1a_2\cdots a_i$ で，y が j 個の記号の列 $y = b_1b_2\cdots b_j$ であるとき，連接 xy は長さ $i + j$ の列になる：$xy = a_1a_2\cdots a_ib_1b_2\cdots b_j$

記号と記号列を区別するための習慣

我々は，記号を表すのにアルファベットの前の方の小文字を使い，列を表すにはアルファベットの後ろの方の文字，特に w, x, y, z を使う．この習慣に慣れると，議論している要素のタイプがすぐわかるようになるであろう．

例 1.25 $x = 01101, y = 110$ とすると，$xy = 01101110$，$yx = 11001101$ となる．任意の列 w について，等式 $\varepsilon w = w\varepsilon = w$ が成り立つ．このように ε は「他のどんな列と連接しても結果はその相手の列になる」ので，連接に対する**単位元**である（似た例に 0 がある：これは他のどんな数 x に加えても答が x になるので，加法についての単位元である）．▨

1.5.3 言　　語

ある特定のアルファベット Σ 上の列，つまり Σ^* から抜き出された列の集合を**言語**という．Σ がアルファベットで $L \subseteq \Sigma^*$ ならば，L を **Σ 上の言語**という．Σ 上の言語は，Σ のすべての記号を含む必要はない．だから L が Σ 上の言語であれば，Σ を含む任意のアルファベット上の言語ともいえる．

「言語」という用語は奇妙に聞こえるかも知れない．しかし，日常の言語は文字列の集合と見ることができる．一つの例が英語で，正しい英単語の集合はすべての英字から成るアルファベット上の列の集合である．もう一つの例は C などのプログラミング言語で，合法的なプログラムはその言語のアルファベットから構成される可能な記号列の部分集合を形作る．そのアルファベットはアスキー（ASCII）記号の部分集合である．細かくいうとプログラミング言語が違えばアルファベットもほんの少し違うが，ふつうは英字の大文字・小文字，数字，句読点，それに数学記号が含まれている．

しかし，オートマトン理論を学ぶと，もっと多くの異なる言語に出会う．次にいくつか例を示そう．

(1) ある $n \geq 0$ について，n 個の 0 のあとに n 個の 1 が並んだ列から成る言語：

$$\{\varepsilon, 01, 0011, 000111, \cdots\}$$

(2) 0 と 1 とが同じ数だけ含まれているような列の集合：

$$\{\varepsilon, 01, 10, 0011, 0101, 1001, \cdots\}$$

(3) 素数を表す 2 進数の集合：

$$\{10, 11, 101, 111, 1011, \cdots\}$$

(4) 任意のアルファベット Σ に対して，Σ^* は言語である．

(5) 空集合 $\varnothing$ は，任意のアルファベット Σ の上の言語である．

(6) 空列だけから成る言語 $\{\varepsilon\}$ も，任意のアルファベット Σ の上の言語で

ある．$\emptyset$ と $\{\varepsilon\}$ とは異なることを注意しておこう：前者は列を何も含まないが，後者は一つの列を含んでいる．

言語についてのただ一つの重要な制約は，そのアルファベットが有限である，ということである．だから言語は，無限個の文字列を含んでよいが，特定の有限アルファベットから作られる文字列から成るものに限られる．

1.5.4 問　　題

オートマトン理論で**問題**とは，与えられた文字列がある特定の集合に属しているか否かを決定することである．あとでわかるように，我々が一般的に「問題」と呼んでいることは，ある言語への帰属問題として表現できる．正確にいうと，Σ があるアルファベットで L が Σ 上の言語であるとき，問題 L とは：

- 与えられた Σ^* の中の文字列 w について，w が L の中にあるか否かを決定せよ．

言語を定義する手段としての集合の記法

言語はふつう「集合の記法」で記述される：

$\{w \,|\, w$ についての条件 $\}$

この表現は,「(タテ棒の右に書いてある条件) を満たす語 w の集合」と読む．次の例を示す：

(1) $\{w \,|\, w$ は同じ個数の 0 と 1 を含む $\}$

(2) $\{w \,|\, w$ は素数を表す 2 進数である $\}$

(3) $\{w \,|\, w$ は文法的に正しい C プログラムである $\}$

w を，変数を含む式に置き換え，その変数についての条件によって，言語の中の文字列を記述することもよくある．次の例では，最初のものは変数 n を含み，2 番目の例は変数 i, j を含んでいる．

(1) $\{0^n1^n \,|\, n \geq 1\}$ これは「1 以上の n に対する，0 の n 乗・1 の n 乗の集合」と読む．この言語は列 01，0011，000111，$\cdots$ から成る．アルファベットの場合と同じように，ある一つの記号の n 個のコピーを並べた列を表すのに，その記号の n 乗といういい方をする．

(2) $\{0^i1^j \,|\, 0 \leq i \leq j\}$ この集合は，いくつかの 0（なくてもよい）のあとにそれ以上の個数の 1 を続けた列から成る．

言語？ 問題？

言語と問題は実は同じものである．どちらの用語で呼ぶかは，視点によってきまる．たとえば $\{0^n1^n \mid n \geq 1\}$ の中の文字列それ自身について考えるときは，その文字列の集合を言語と考える方に傾く．この本の最後の章では，記号列に「意味」を与える：たとえば記号列がグラフ，論理式，あるいは整数を表す符号とみなす．そのように，文字列それ自身よりも文字列が表しているものに関心がある場合には，文字列の集合を問題と考えることが多い．

例 1.26 素数判定の問題は，その値が素数であるような 2 進数全体の集合言語 L_p で表現できる．すなわち，与えられた 0 と 1 の列に対して，それがある素数を表す 2 進数であるとき「はい」と答え，そうでなければ「いいえ」と答えよ．ある列に対してこの決定は容易である．たとえば 0011101 は,「0 を除くどんな整数の 2 進表現も 1 で始まる」という単純な理由で,「素数を表す 2 進数」ではない．しかし 11101 が L_p に属するか否かは，それほど明らかでない．この問題を解くには，時間か記憶容量か，あるいはその両方の計算資源をかなりの程度使わざるを得ないであろう．▨

上のような「問題」の定義には，不満足な面があり得る：多くの人は問題を決定問題「以下のことは正しいかどうか？」ではなく，何かの入力に計算あるいは変換を施す要求「その任務を果たす最善の方法を見いだせ」と考える．たとえば，C コンパイラのパーサーの任務，すなわち「与えられたアスキー記号の列が正しい C プログラムの集合 L_c に属しているかどうかを判定する」という要求は，我々の形式的な「問題」の定義に適合しているが，パーサーはその判定をするだけではなく，構文木や記号表の項目など，いろいろなものを作り出す．なお悪いことに，コンパイラ全体としては C プログラムをある機械のためのオブジェクトコードに変換する問題を解決してくれるので，プログラムの文法的正しさについて「はい」とか「いいえ」と答えるだけではまったくない．

しかし,「問題」を言語と定義することは，計算量の理論で重要な問題を扱う妥当な方法として，時間の検証にも耐えてきた．この理論では，ある問題の計算量の下限を証明することに興味がある．特に重要なのは，ある問題が入力の指数関数よりも少ない時間量で問題が解けないことを証明するための技法である．はい／いいえを答えるという言語型の問題は,「これを解け」という意味で

の問題と，よく知られている多くの場合に「同程度にむずかしい」ことがわかっている．

すなわち，与えられた文字列がプログラミング言語 X の正しい記号列から成る言語 L_X に属しているかどうかの決定がむずかしいことが証明できるときは，その言語 X のプログラムをオブジェクトコードに翻訳することは，よりやさしくはないであろう．コードを生成するのがやさしいとすれば，翻訳プログラムを走らせてオブジェクトコードの作成に成功するかどうかを見れば，入力が L_X の正規のメンバーであるかどうかが判定できるからである．オブジェクトコードが生成されるかどうかの判定は容易であるから，オブジェクトコードを生成する高速アルゴリズムを，帰属性の判定に有効に利用できる．だから「L_X の帰属性判定はむずかしい」と仮定すると，矛盾が出る．このように「もしこの帰属性判定がむずかしければ，プログラミング言語 X で書かれたプログラムのコンパイルもむずかしい」という主張が背理法によって証明できた．

ここではある効率のよいアルゴリズムがあると仮定して，それがむずかしいとすでにわかっている別の問題にも効率よく使えることを利用して，ある問題が「むずかしい」ことを証明したが，この技法は第2の問題から最初の問題への「還元」(reduction) と呼ばれる．これは計算量の問題の研究では重要な手段で，問題を一般的な問から「帰属問題」に限定した我々の概念を大いに支持してくれる．

1.6 第1章の要約

- **有限オートマトン**：有限オートマトンは状態と入力への応答としての状態変化にかかわっている．これはコンパイラの字句解析部分や，回路あるいはプロトコルが正しいかどうかを検証するシステムなど，何種類かのソフトウェアの構築に役立つ．
- **正則表現**：これは有限オートマトンで表現できるのと同じ文字列パターンを記述する構造的な記法である．たとえばテキストやファイル名の中からあるパターンを検索する道具など，多くの一般的なソフトウェアで使われている．
- **文脈自由言語**：これは構造的なプログラミング言語とそれに関連する文字列の集合を記述する重要な記法である．コンパイラのパーサー部分を構築するときにも役立つ．
- **チューリング機械**：現実のコンピュータの能力をモデル化したオートマトンである．これによって決定問題や，コンピュータに何ができ何ができないかの研究が可能に

なった．これはまた，何とか手に負える問題（多項式時間で解ける問題）と手に負えない問題（それでは解けない問題）とを区別するのに役だった．

◆**演繹的証明**：証明は基本的には文のリスト（列）である：どの文も，正しいと仮定されているか，またはそれより前の文から論理的に導かれるものでなければならない．

◆**「もし … ならば」文の証明**：多くの定理は「もし…ならば…である」という形をしている．「もし」に続く文（群）が仮定，「ならば」のあとが結論である．「もし…ならば」文の演繹的証明は，仮定に始まり，仮定とそれ以前の文から論理的に導かれる文が並び，最後は（それらの文の一つとして）証明すべき結論で終わる．

◆**必要十分条件文の証明**：「…であるための必要十分条件は…である」という形の定理もある．これは両方向の「もし…ならば」文を示すことによって証明される．二つの異なる形で記述された集合の一致を主張する定理も，同類である；一方が他方に含まれていることを，両方向で示すことによって証明される．

◆**対偶の証明**：「もし H ならば C」という文を証明したいとき，「もし C でなければ H でない」ことのほうが証明しやすいことがある．後者を前者の対偶という．

◆**背理法**：「もし H ならば C」を証明するのに，「もし H でしかも C でないとすると，…（何か誤りとわかっていること）」を証明するほうがラクなこともある．この型の証明は背理法と呼ばれる．

◆**反例**：ある文が正しくないことを示したいときもある．もしその文がいくつかの変数を含んでいるなら，それが正しくないことを証明するには，ただ一つの反例——その文が偽になるような変数の与え方を示すだけでよい．

◆**帰納的証明**：整数変数 n を含む文は，n についての帰納法で証明されることが多い．まず「基礎」として，n のいくつか有限個の値について，文が正しいことを証明する．それから「帰納的ステップ」として，その文が n 以下の値について正しければ，$n+1$ についても正しいことを証明する．

◆**構造的帰納法**：ある状況では，本書でもよく現れるが，証明すべき定理はある再帰的に定義された構造（たとえば木）にかかわっている．そのような定理は，かかわっている対象を「組み立てるまでのステップ数」についての帰納法によって証明できることがある．

◆**アルファベット**：アルファベットとは任意の記号の有限集合をいう．

◆**文字列**：文字列（列，記号列ともいう）とは記号を有限個並べた列のことである．

◆**言語と問題**：言語とは，ある特定のアルファベットから記号を選んで作られた文字列の集合のことである（無限集合でもよい）．ある言語 L の中の文字列がある仕方で解釈されるとき，「ある列がその言語に属しているかどうか」は問題 $<L$ にかかわる帰属性問題，問題 $L>$ と呼ばれることがある．

1.7 第1章の参考書

この章の材料を，コンピュータ科学の基礎にある数学的な諸概念を含めて幅広く解説している本として，次の［1］を勧める．

[1] A. V. Aho and J. D. Ullman, "*Foundations of Computer Science*," Computer Science Press, New York, 1994.

2 有限オートマトン

この章では「正則言語」と呼ばれる言語の集まり（クラス）を紹介する．先に 1.1.1 項で簡単な有限オートマトンの例を紹介したが，正則言語の全体は有限オートマトンで記述できる言語の全体と一致する．以下では，はじめにある程度大きな例を用いて研究の動機を説明し，その後で有限オートマトンの正確な定義を述べる．

前にも述べたが，有限オートマトンはいくつかの状態を持ち，外界からの「入力」に応じて一つの状態から次の状態へと「制御」が移動する．有限オートマトンはいくつかの種類に分けられるが，その際注目する点の一つは，オートマトンの制御が「決定性」（同時に二つ以上の状態にいることはできない）か，それとも「非決定性」（同時にいくつかの状態にいることができる）かである．以下で我々は，決定性の有限オートマトンと非決定性のものとで記述できる言語に本質的な差はないことを学ぶが，具体的な応用の際には，非決定性オートマトンの方が決定性よりはるかに書きやすいことに気付くだろう．実際，非決定性の場合は問題の答を直接書く代わりに，ある種の高級言語を使って「プログラム」することが認められているようなもので，それを本章で述べるアルゴリズムによって決定性有限オートマトンに「コンパイル」すると，通常のコンピュータ上で「実行」できる形になる．

この章の最後の部分で，決定性有限オートマトンをさらに拡張して，ある状態から何も入力を読まずに（つまり,「入力」として空列を受け取って）別の状態に移ることもできるような体系を取り上げ，そのように拡張したオートマトンでもやはり正則言語以外のものは記述できないことを示す．なお，この拡張は 3 章で正則言語とオートマトンの等価性を示す際に重要な役割を演じる．

正則言語に関する考察は 3 章でも引き続き行う．そこでは，正則言語を記述するもう一つの方法である正則表現という一種の代数的記法を導入する．そして，正則表現の基本的な性質を調べ，それが有限オートマトンと同等であるこ

とを見たあと，さらに4章で正則言語の重要な性質をオートマトンと正則表現を使って調べる．たとえば，ある言語が正則言語であることを，他の言語が正則であることを使って間接的に導くのに有用な「閉包性」と呼ばれる性質や，決定問題に関する性質などがこれに当たる．決定問題というのはたとえば，二つのオートマトンや正則表現が与えられたとき，それらが同じ言語を表すかどうかという種類の問題で，それらに答えるために具体的なアルゴリズムを示す．

2.1 有限オートマトンの直観的な説明

この節では，現実的な問題に関する多少複雑な例を取り上げ，その問題を解く上で有限オートマトンがどのような役割を果たすかについて考える．インターネット上で買い物をする人が支払いのために使う「電子マネー」── つまり，売り手（店）が実際の貨幣と見なして受け取ることのできるファイル ── のためのプロトコルについて考える．その際，店はそのファイルがにせものやコピーでないことを知る必要があり，買い手（客）はそのファイルのコピーを手許に残しておく必要がある．

そのファイルがにせものでないことは，銀行と暗号制度によって保証されるべきである．すなわち，銀行という第三者がそのファイルを発行し暗号化することによって，にせものかどうかという問題を解決することができる．しかし，銀行のすべき仕事にはもう一つ重要なものがある．その銀行が発行したすべての正当な電子マネーのデータベースを保持し，店が受け取ったファイルは確かに本物の貨幣を代表するものであって，店の口座にその代価が確実に入金されることを保証することである．この問題の暗号理論的側面，および銀行が何十億ドルという電子マネーをどのようにして蓄えたり回収したりするかという問題にここでは立ち入らない．この種の問題が長期にわたって電子マネーに対する重要な障害になるとは予想し難いし，小規模な電子マネーは1990年代後半から実際に少しずつ使われ始めている．

しかし，電子マネーを円滑に機能させるには，人々が希望する多様な方法で電子マネーを扱えるように，よく考えられたプロトコルを作る必要がある．貨幣による取引きには常に不正が付き物であるから，どんな方式を採用するにせよ，その方式のもとで貨幣がどのように使用されるかよく確かめる必要がある．すなわち，起こり得ることはすべて予め想定されたことのみで，想定外のユー

ザが誰かのお金を盗んだり偽造貨幣を使ったりすることは絶対に起こり得ないことを検証する必要がある．この節では以後，ある簡単な（問題点を含んでいる）電子マネーのプロトコルの例を示し，それを有限オートマトンを用いて表したのち，オートマトンに対するある種の変形操作により，そのプロトコルの正しさ（またはこの例の場合のように，プロトコルに欠陥があること）がどのようにして調べられるかを見ることにしよう．

2.1.1 基本的な規則

このプロトコルの登場人物は，客と店と銀行の 3 人である．簡単のためここでは一つのマネーファイル（電子マネー）だけについて考える．客はこのファイルを店に送ることができる．すると，店はそのファイルを銀行で兌換し（すなわち，客の所有ではなく店の所有する新しいマネーファイルを銀行に発行してもらい），商品を客のもとに発送する．一方，客はここでそのファイルをキャンセルすることも可能である．すなわち，客はその金を支払い停止にして客自身の口座に戻すよう銀行に依頼することができる．このように，3 人の登場人物が取り得る行動は以下の五つに限られる．

(1) 客が支払いのため電子マネーを店に送る．

(2) 客が電子マネーの取り消しを行う．すなわち，客が電子マネーを銀行に送り，その金額を客の口座に加えるよう要求する．

(3) 店が商品を客に発送する．

(4) 店が電子マネーを兌換する．すなわち，電子マネーを銀行に送り，その金額を店の口座に加えるよう要求する．

(5) 銀行が暗号化された電子マネーを新たに作り，それを店に転送する．

2.1.2 プロトコル

上のプロトコルで，登場する 3 人の行動を注意深く設計しないとおかしなことが起きる可能性がある．次に述べる我々のプロトコルでは，客は常に責任ある行動をするとは限らないという当然の前提をおく．特に，客は電子マネーをコピーして何度も使おうとするかも知れないし，あるいは，電子マネーの支払いと同時に取り消しを行い品物をタダで手に入れようとするかも知れない．

一方，銀行は責任ある行動をしなければ銀行とはいえない．特に，二つの商店が同じ電子マネーを同時に兌換することがあってはならないし，また，電子

マネーが取り消しと同時に兌換されることがあってはならない．商店も商店としての責任において行動しなければならない．特に，商品に対する代価が確実に支払われたことを確認した後でなければ商品を発送してはならない．

この種のプロトコルは有限オートマトンを使って表すことができる．オートマトンの各状態は，3人の登場人物がどんな状況にいるかを表す．言い換えると，それらの状態は，重要な事象のいくつかが起き，その他の事象はまだ起きていないことを「記憶」するものである．そして，オートマトンの状態が変わるのは上に述べた五つの事象のうちの一つが起きたときである．このようにして我々は3人の登場人物を三つのオートマトンによって表し，上に述べた五つの事象は（実際には3人の登場人物の誰かによって引き起こされるのだが），オートマトンの「外界」にあるものとして扱う．その理由は，今の場合重要なのは，誰がどの事象を引き起こすかではなく，全体としてどんな順序でどの事象が起き得るかだからである．

図2.1に，3人の登場人物を表すオートマトンを図示する．この図では登場する3者に直接関係のある事象のみを示す．たとえば，支払いは客と店のみに関係する．銀行はその事象には感知せず，あとに店が兌換を実行した時点ではじめてその事実を知る．

はじめに，銀行を表すオートマトン (c) について説明する．その開始状態は状態1で，これは銀行がある電子マネーを発行し，それに対する兌換の要求も取り消しの要求も起きていない状況を表す．もしここで客から銀行へ取り消しの要求があれば，銀行はその金額を客の口座に戻し，状態2へ移る．この状態2は，その電子マネーが取り消されたことを表す状態である．銀行は，一旦状態2に入るとそこに留まらなければならない．なぜなら，銀行は同じ電子マネーをもう一度取り消したり，客が再度それを使ったりすることは責任上認められないからである[1]．

---

[1] ここでの議論は単独の電子マネー（のファイル）についての話であることに注意せよ．実際には，銀行は同じプロトコルで沢山の電子マネーを扱うが，それらのプロトコルの働きは同じで互いに干渉し合わないから，我々はあたかも一つの電子マネーだけがあるかのように考えることができる．

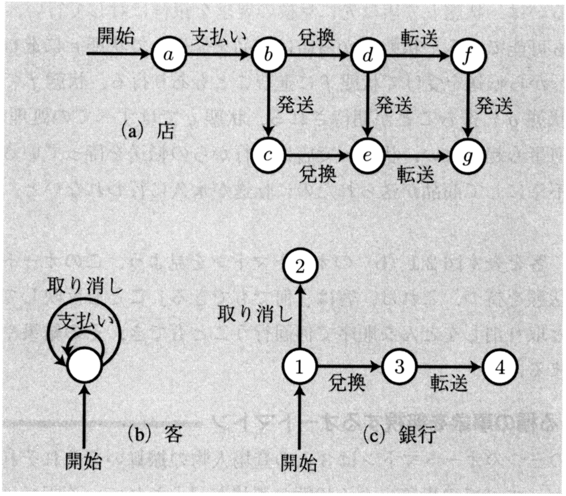


図 2.1 客と店と銀行をそれぞれ表す三つの有限オートマトン

次に，銀行が状態 1 で店から兌換の要求を受けた場合について考える．そのとき，このオートマトン (c) は状態を 3 に変え，次いで店名義の新しい電子マネーを発行し，転送のメッセージと共にそれを店に送る．そして銀行は状態 4 に移る．この状態 4 で銀行はこの電子マネーに関する取り消しや兌換やその他一切の要求を受けつけない．

次に，店の行動を表すオートマトン図 2.1 (a) について考えよう．銀行は常に正しい行動をするが，店の行動には欠陥がある．商品の発送と経理操作が別々に行われ，したがって商品の発送が電子マネーの兌換の前か，後か，あるいはその最中に行われる場合を想像してほしい．その場合，このオートマトンに示されたやり方では，店が商品を発送した後で電子マネーがにせものだとわかることがあり得る．

店の開始状態は a である．客が支払いを実行して商品を発注すると，店は状態 b に移る．この状態では，店は商品の発送と電子マネーの兌換を行うことができる．もし先に商品を発送すると，店の状態は c に移る．この状態で店は電子マネーの兌換を銀行に要求し，銀行からその金額の転送を受けなければなら

ない．あるいは，状態bで店は先に兌換の要求を銀行に対して行い，状態dに移ることも可能である．状態dから店は商品を発送して状態eに進むか，または先に銀行から転送を受けて状態fに進むこともあり得る．状態fで店は商品を発送し状態gに進むことが期待される．状態gではすべての処理が終わり，それ以後何事も起きない．状態eで店は銀行からの転送を待っている状態である．もし不幸にして商品が送られたのに転送が永久に行われないと，店は損害を被る．

最後に，客を表す図2.1（b）のオートマトンを見よう．このオートマトンは唯一つの状態を持つ．これは，客は「何でもできる」ことを反映している．客は，支払と取り消しをどんな順序で何回行うこともでき，その結果やはり同じ状態に留まる．

2.1.3 ある種の事象を無視するオートマトン

図2.1の三つのオートマトンは3人の登場人物の振舞いをそれぞれ独立に表しているが，すべての事象による状態の遷移が記入されている訳ではない．たとえば，店は電子マネーの取り消しのメッセージには直接影響されないから，たとえ客によって取り消しが行われても，店の状態は変化しない．しかし，(2.2節で述べる）有限オートマトンの正式な定義では，オートマトンはどんな入力Xを受け取っても，現在いる状態からXというラベルのついた辺に沿って状態を移動することになっている．その約束にしたがえば，店を表すオートマトンに，各状態から自分自身に向かう辺で取り消しというラベルのついたものを新しく書き加えなければならない．そうすると，取り消しが実行されたとき店のオートマトンは，取り消しという入力に対して前と同じ状態に留まるという「状態遷移」を行うことになる．今述べた新しい辺をもし加えないと，取り消しが行われたとき店のオートマトンは「死ぬ」ことになる（すなわち，そのオートマトンはどの状態にもいられないことになってその後の行動は一切不可能になる）．

上のオートマトンに関する別の問題点として，3者のうちの誰かが，意図的にか誤ってか，予想外のメッセージを送った場合，その行動がどれかのオートマトンを死に至らしめる事態を引き起こすかも知れないことである．たとえば，店がオートマトン（a）の状態eにいるとき，客が2度目の支払いを行ったとす

る．すると，状態 e から支払いと書かれた辺は出ていないため，店のオートマトンは銀行から転送を受ける前に「死ぬ」ことになる．この事態を避けるために，図 2.1 のオートマトンのある状態に，その状態にいるとき無視すべき事象をラベルに持つループ（ある状態から自分自身に向かう辺）を付け加える必要がある．そのようにして得られた完全なオートマトンを図 2.2 に示す．スペースの節約のため，出発点と終着点が共通でラベルのみが異なる複数の辺があるとき，一つの辺に複数のラベルを書き込むという便法を用いた．ここで新たに加えた事象には次の二つの種類のものがある．

(1)　そのオートマトンが表現している登場人物に無関係な事象．店にとって無関係な事象は，前にも触れた取り消しだけである．そこで，オートマトン（a）のすべての状態に取り消しをラベルに持つループを加える．銀行

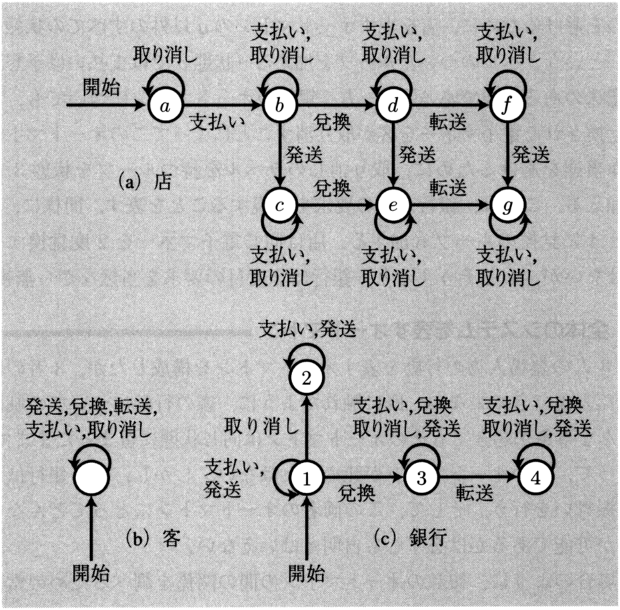


図 2.2　完全な状態遷移を持つ三つのオートマトン

にとって無関係な事象は，支払いと商品の発送であるから，オートマトン (c) のすべての状態にその二つのラベルを持つループを加える．客にとって商品の発送と兌換と転送は無関係な事象だから，オートマトン (b) のすべての状態にその三つのラベルを持つループを加える．すると結局このオートマトン (b) は，すべての事象に対してその唯一の状態に留まることになり，客の状況は全体のシステムの中で何らの効果も発揮しないことになる．ただし，それでも客は登場人物の一人としてこのシステムに参加し，支払いと取り消しを行う当事者であることに変わりはない．しかし前に述べた通り，誰がそれぞれの事象を行うかを，これらのオートマトンが問題にしているのではない．

(2) オートマトンが「死ぬ」のを避けるために加えた事象．先に言及したとおり，客が支払いを2度実行することにより店のオートマトンが「死ぬ」のを避けるために，店を表すオートマトンの a 以外のすべての状態に支払いというラベルのついたループを加える（状態 a では支払いは予想された意味のある事象である）．一方，銀行のオートマトンについても，すでに兌換された電子マネーを客が取り消すことによってこのオートマトンが死ぬ事態を避けるために，取り消しのラベルを持つループを状態3と4に加える．これは，銀行がその要求を無視することを表す．同様に，状態3と4に兌換のループも加える．店は同じ電子マネーを2度兌換すべきではないが，もしそうすると，銀行は2度目の要求を当然ながら無視する．

2.1.4 全体のシステムを表すオートマトン

上で3人の登場人物の行動を表すオートマトンを構成したが，3者の間の関係はまだ表現できていない．既に触れたように，客の行動についての制限はなく，どんな事象が起きてもそのオートマトンは同じ状態に留まったままである．したがって，このオートマトンが死ぬことはない．しかし，店と銀行はもっと複雑な振舞いを行う．そして，この両者のオートマトンにとってどんな状態の組合せが可能であるかは必ずしも自明とはいえない．

この場合のように，複数のオートマトンの間の関係を調べるための常套手段は，**積**オートマトンを構成することである．積オートマトンの状態は，片方のオートマトンの状態ともう一方のオートマトンの状態の対として構成される．

たとえば，積オートマトンの状態 $(3, d)$ は，銀行のオートマトンの状態が 3 で，店のオートマトンの状態が d であることを表す．銀行のオートマトンは四つの状態を持ち，店の方は七つの状態を持つから，したがってその両者の積オートマトンは全部で $4 \times 7 = 28$ 個の状態を持つ．

図 2.3 に，その積オートマトンを，わかりやすいように 28 個の状態を縦横に整列させて示す．ただし，四つの行はそれぞれ銀行のオートマトンの状態に対応し，七つの列はそれぞれ店のオートマトンの状態に対応している．また，紙面の節約のため，辺につけるラベル，支払い（pay），発送（ship），取り消し（cancel），兌換（redeem），転送（transfer）をそれぞれ短縮形 P, S, C, R, T で表す．

積オートマトンの状態間を結ぶ辺は，銀行と店のオートマトンを「同時に」運行させたときの振舞いを表す．積オートマトンの状態の第 1 要素と第 2 要素は互いに独立に様々の入力に反応して遷移を行う．しかし注意すべきことは，もし与えられた入力に対して，積オートマトンのもとになった二つのオートマトンのうちのどちらかで可能な状態遷移を提示していないならば，積オートマトンも行き先がない（したがってそのオートマトンは死ななければならない）．

上に述べた状態遷移を正確に構成する方法は次の通りである．今，積オートマトンの状態が (i, x) で，入力事象が Z の場合を考える．この状態 (i, x) は，銀行が状態 i で，店が状態 x であることを表す．まず銀行のオートマトンで，状態 i からラベル Z を持つ遷移の有無を見る．今，そのような遷移があり，その行く先が状態 j だとする（もしその遷移が自分自身に戻るループならば，j は i 自身である）．次に，店のオートマトンにおいて，x からラベル Z を持つ辺がどこかの状態 y に向かっているかどうかを見る．もしそのような j と y があれば，積オートマトン上で，(i, x) から (j, y) へ向かうラベル Z を持った辺を書く．もしそのような j か y のどちらか一方でも存在しない（すなわち，銀行か店のオートマトンで i または x からラベル Z を持つ辺が出ていない）ならば，積オートマトンで，(i, x) からラベル Z を持つ辺はない．

図 2.3 の辺はそのようにして構成したものである．たとえば，入力 P に対して，店は a から b へ進むが，a 以外の状態ではその状態のまま動かない．銀行は，入力 P とは無関係なため，この入力に対して今いる状態のまま動かない．この事実から，図 2.3 の四つの行の左端の状態から出るラベル P を持つ四つの

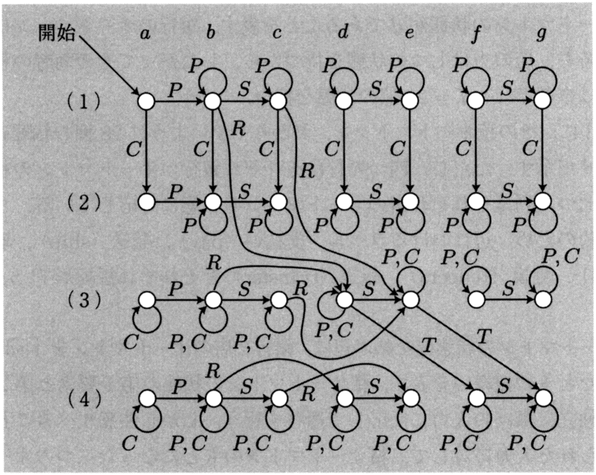


図 2.3 店と銀行に対する積オートマトン

辺と，その他の状態におけるラベル P を持つループの説明がつく．

別の例として，入力 R について見ると，銀行が状態 1 で兌換 R のメッセージを受け取ると，状態 3 に進む．状態 3 か 4 でそのメッセージを受けると，その状態に留まるが，状態 2 だと銀行のオートマトンは死ぬ．すなわち，行き先がない．一方，店は，兌換 R のメッセージを受けると，状態 b から d へ，また c から e へ進む．図 2.3 には R をラベルに持つ辺が六つあるが，それらは銀行のオートマトンにおける R をラベルに持つ三つの辺と店のオートマトンにおける同じく R をラベルに持つ二つの辺の組合せに基づくものである．たとえば，兌換 R は銀行を状態 1 から 3 に，店を b から d に移すから，積オートマトンで状態 $(1, b)$ から $(3, d)$ へラベル R を持つ辺を書く．同じく，兌換 R は銀行を状態 4 から 4 に，店を c から e に移すから，積オートマトンで状態 $(4, c)$ から $(4, e)$ へラベル R を持つ辺を書く．

2.1.5 積オートマトンによるプロトコルの確認

図 2.3 から次のことがわかる．28 個の状態のうち開始状態 $(1, a)$——銀行のオートマトンの開始状態 1 と店のオートマトンの開始状態 a に対応する状態——から到達可能なのは 10 個の状態だけである．たとえば，状態 $(2, e)$ や $(4, d)$ に開始状態 $(1, a)$ から到達する経路はないため，これらの状態は到達可能ではない．このように到達不可能な状態をオートマトンに含める必要はないが，この図では系統的に構成法を示す目的でそれらを含めて書いた．

しかし，この例のようにオートマトンを使ってプロトコルを解析する本当の目的は,「$\cdots$ のようなタイプのエラーは起きる可能性があるか?」という種類の質問に答えることである．たとえば今の例の場合，店が商品を発送したのに代金を受け取れないことがあるか，という問が考えられる．すなわち，積オートマトンは，店が商品を発送した状態（つまり，c, e, g のいずれかの列）で，かつ入力 T による遷移が（これまでも今後も）行われないことがあるか，という問である．

たとえば，状態 $(3, e)$ では，商品は既に発送され，やがて入力 T による遷移が行われて状態 $(4, g)$ に到達するはずである．銀行に関して，状態 3 に到達したということは，兌換 R が要求されそれが処理されたことを意味する．さらに，その兌換の要求を受ける前に状態 1 で取り消し C を受けなかったことと，今後もしそれを受け取ったとしても無視する等である．したがって，銀行はやがて店に電子マネーの転送をすることは確実である．

しかし，状態 $(2, c)$ には問題がある．まず，この状態は開始状態から到達可能である．しかし，この状態を出て進む先は自分自身のみである．この状態は，銀行が（店からの）兌換の要求の前に（客から）取り消しの要求を受けた状況を表す．しかし，店は（客から）支払いの通知を受けている．つまり，客は二枚舌を使って，同じ電子マネーの支払いと取り消しとを要求している．一方，店は愚かにもその電子マネーの兌換の要求をする前に商品を発送した．そして後に店が兌換の手続きをとると，銀行はその手続きを受け取ったという通知さへ出さないだろう．なぜなら，銀行は状態 2 にいて，2 という状態では銀行はその電子マネーを取り消し，兌換の要求を受け付ける状態ではないからである．

2.2 決定性有限オートマトン

この節では，有限オートマトンの正式な定義を述べる．それによってこれまで 1.1.1 項や 2.1 節で直観的に説明していたものを厳密に表現できるようになる．はじめに，決定性有限オートマトンの定義を述べる．これは，どんな入力列を読んだ場合もその状態は一意的に定まるという性質を持つオートマトンである．「決定性」ということばは，どんな入力記号についても，現在の状態からある一つの状態に遷移するオートマトンという意味であり，これに対して 2.3 節で述べる「非決定性」有限オートマトンの場合は，その時点で可能ないくつかの状態が指定される．以後，単に「有限オートマトン」というと決定性有限オートマトンを指す．しかし混乱を避けるために，なるべく「決定性」ということばを前に付けるか，または決定性有限オートマトン (deterministic finite automaton) の頭文字をとって **DFA** と呼ぶことにする．

2.2.1 決定性有限オートマトンの定義

決定性有限オートマトンは次の要素からなる．

(1) **状態**（state）の有限集合．ふつう Q と書く．

(2) **入力記号**（input symbols）の有限集合．ふつう Σ と書く．

(3) 状態と入力記号を与えると状態を返す**遷移関数**（transition function）．ふつう δ と書く．先にオートマトンを直観的に図で示したとき我々は状態遷移関数 δ を，状態間のラベル付き辺によって表した[2)]．つまり，q から p へラベル a のついた矢印があるとき $\delta(q,a)$ の値は p である．

(4) **開始状態**（start state）．これは Q の一つの要素である．

(5) **最終状態**（final state）または**受理状態**（accepting state）の集合．これは Q の部分集合で，ふつう F と書く．

前にも述べたように，決定性有限オートマトンを略して（英語名 deterministic finite automaton の頭文字をとって）DFA と呼ぶが，DFA を表す最も簡単な方法は上の五つの要素を列挙することである．証明などで具体的にある DFA を提示する際，我々は次のような「5 個組」の記法を用いる．

---

2) より正確にいうと，遷移関数 δ はグラフで表され，そのグラフ中の辺は遷移関数によって表される個々の状態遷移を表す．

$$A = (Q, \Sigma, \delta, q_0, F)$$

ここで，A は DFA の名前であり，Q はその状態の集合，Σ はその入力記号の集合，δ はその遷移関数，q_0 はその開始状態，F はその受理状態の集合をそれぞれ表す．

2.2.2 DFA はどのように列を処理するか

DFA について最初に理解すべきことは，DFA が与えられた入力記号列を「受理」するか否かをどのようにして決めるか，その方法を知ることである．DFA $A = (Q, \Sigma, \delta, q_0, F)$ に対して $a_1a_2\ldots a_n$ を入力記号列として与えるとする．DFA が仕事を始めるときは必ずまずその状態を開始状態 q_0 にセットする．そして，最初の入力記号 a_1 によって A がどの状態に進むかを知るには，遷移関数 δ を使って $\delta(q_0, a_1)$ を調べればよい．今 $\delta(q_0, a_1) = q_1$ だとしよう．入力列の次の記号 a_2 を読み込んだときの状態は $\delta(q_1, a_2)$ となる．この状態を q_2 とする．以後同様にして $\delta(q_{i-1}, a_i) = q_i$ の関係にある状態 $q_3, q_4, \ldots, q_n$ を見付ける．このとき，もし q_n が F の要素 (すなわち，受理状態) であれば，入力列 $a_1a_2\ldots a_n$ は受理され，そうでなければ「拒否」される．このようにして DFA A が受理する記号列の全体を A の「言語」と呼ぶ．

例 2.1 0 と 1 から成る列で，その中に 01 がどこかに現れるもののすべてを，そしてそれのみを受理する DFA を上の定義にしたがって構成してみよう．この言語 L は次のように表される．

$$\{\, w \mid w \text{ は } x01y \text{ の形の列，ただし } x, y \text{ は } 0 \text{ と } 1 \text{ から成る適当な列} \,\}$$

これと同じことを縦棒の左側に x と y を入れて書くと

$$\{\, x01y \mid x \text{ と } y \text{ は } 0 \text{ と } 1 \text{ から成る列} \,\}$$

となる．この言語に属する列の例として $01, 11010, 100011$ があり，この言語に属さない列の例としては，$\varepsilon, 0, 111000$ がある．

この言語を受理するオートマトンを作るにはどうすればよいだろうか? まず，入力アルファベットは $\Sigma = \{0, 1\}$ である．状態の集合 Q は少なくとも開始状態 q_0 を含む．このオートマトンは入力列のすでに読んだ部分について重要ないくつかの事柄を記憶する必要がある．実際，A は入力列の中に部分列として 01 が含まれているかどうかを決定するために，次のことを記憶しておく必要がある．

(1) すでに 01 に出会ったか? もしそうなら，以後どんな入力記号を読んでも受理する．すなわち，以後は受理状態にとどまる．

(2) まだ 01 に出会っていないが直前に読んだ記号が 0 のとき，もし次に 1 を読んだら，01 を続けて読むことになるからその後何を読んでも受理してよい．

(3) まだ 01 に出会わず，直前に読んだ記号は 0 でない（つまり，まだ一つも記号を読んでいないか，または直前に読んだ記号が 1 である）とき，A は，今後 0 を読んだ直後に 1 を読むまで入力を受理しない．

これらの三つの条件はそれぞれ一つの状態として表現できる．条件（3）は開始状態 q_0 によって表現される．実際，オートマトンが動作を開始したばかりのときは，その後 0 を読んだ直後に 1 を読むまで入力を受理しない．そして，もし開始状態 q_0 で 1 を読んだら q_0 にとどまる．その理由は，01 を見るというゴールに関して，開始状態のときより近づいてはいないからである．これを式で書くと $\delta(q_0, 1) = q_0$ となる．

しかし，開始状態 q_0 で 0 を読んだ場合は，条件（2）の状況にいることになる．すなわち，まだ 01 に出会っていないが直前に読んだ記号は 0 である．条件（2）の表す状態を q_2 と呼ぶことにしよう．すると，q_0 で入力 0 を読んだときの状態遷移は式で $\delta(q_0, 0) = q_2$ と表される．

次に，状態 q_2 からの遷移について考える．もしそこで入力記号 0 を読んだら，現在よりも状況がよくなったとはいえないし，悪くなったともいえない．我々はまだ 01 に出会っていないし，0 を直前に読み 1 を待っているという点で，前と同じである．状態 q_2 はまさにそうした状態を表すものである．したがって，これを式で書くと $\delta(q_2, 0) = q_2$ となる．ところが，状態 q_2 で 1 を読むと，0 の直後に 1 がきたのだから受理状態に行くべきである．それは条件（1）に対応する状態である．その状態を q_1 と呼ぶことにする．そのとき，上で述べた遷移は $\delta(q_2, 1) = q_1$ となる．

最後に，状態 q_1 に対する遷移を考える．q_1 はすでに 01 を見た状態だから，今後何が入力されても 01 を見たことに変わりはない．したがって，その状態遷移は $\delta(q_1, 0) = \delta(q_1, 1) = q_1$ となる．

以上をまとめると，$Q = \{q_0, q_1, q_2\}$ で，q_0 が開始状態であり，q_1 が唯一の

受理状態，すなわち，$F = \{q_1\}$ である．こうして，01 を部分列として持つ列全体から成る言語 L を受理するオートマトン A の完全な記述は

$$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

で与えられる．ここで δ は上で定めた関数である． ▨

2.2.3 DFA に関する記法

DFA を 5 個組として記述し，その遷移関数 δ を詳細に記述するのは手間がかかる上に読み難くもある．オートマトンのより簡便な記述法として次の二つの方法がある．

1. 遷移図（2.1 節で示したグラフ）
2. 遷移表（遷移関数の表による表現．これは状態の集合と入力アルファベットの情報を同時に含む．）

遷移図　次のグラフを DFA $A = (Q, \Sigma, \delta, q_0, F)$ の遷移図（transition diagram）という．

a) Q の各状態にグラフの頂点を対応させる．

b) Q に属する状態 q と Σ に属する入力記号 a に対して，$\delta(q, a) = p$ のとき，遷移図の頂点 q から p にラベル a を持つ辺を書く．

c) 開始状態 q_0 に向かう矢印を書き，**開始**というラベルを付ける．

d) 受理状態（すなわち，F の要素）に対応する頂点は二重の輪で表す．F に属さない各状態は一重の輪で表す．

例 2.2　図 2.4 は，例 2.1 で我々が設計した DFA を遷移図で示したものである．この図は三つの状態にそれぞれ対応する三つの頂点があり,「開始」の矢印が向かっているのが開始状態 q_0 で，二重の輪がある q_1 が受理状態である．各状態からラベル 0 の付いた辺が一つと，ラベル 1 の付いた辺が一つ出ている（q_1 の場合は，2 本の辺が一つにまとめられ二つのラベルが付いている）．これらの各辺は例 2.1 で説明したそれぞれの遷移に対応している． ▨

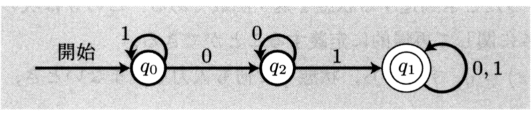


図 2.4　01 を部分列として含む列全体を受理する遷移図

遷移表　遷移表（transition table）は，二つの引数に対して値を返す関数を表形式で示す伝統的な方法で δ を示したものである．ただし，表の各行は状態に対応し，各列は入力記号に対応する．状態 q に対応する行と入力記号 a に対応する列の交点に状態 $\delta(q,a)$ を記入する．

例 2.3　例 2.1 の δ に対応する遷移表を図 2.5 に示す．図の遷移表には開始状態を示す「開始」の矢印 → と受理状態を示す星印 $*$ も併せて記載してある．この図で，DFA の状態の集合と入力記号の集合に関する情報もそれぞれ表の行と列の見出しで示されているから，遷移表によって有限オートマトンは完全に曖昧さなく表される．　▨

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

図 2.5　例 2.1 の DFA に対する遷移表

2.2.4　遷移関数の拡張

DFA が一つ与えられるとそれによって一つの言語（つまり，DFA を開始状態から受理状態に導くような入力記号列の全体）が定義できることを，章のはじめに簡単に述べた．遷移図に即していうと，DFA の言語とは，開始状態から出発して受理状態に至る経路（パス）のラベルを繋いでできる記号列の全体に他ならない．

ところで，DFA の言語という概念を正確に述べるにはどうすればよいだろうか．そのためには，任意の状態から出発して入力列を読みながら進むとどの状態に到達するかを表す関数（それを**拡張された遷移関数**と呼ぶ）を定義すればよい．今，δ が DFA A の遷移関数のとき，その拡張を $\hat{\delta}$ とすると，$\hat{\delta}$ は各状態 q と各入力記号列 w に対してオートマトン A が q から出発して入力列 w を読み進んで行ったとき到達する状態を返す関数である．この $\hat{\delta}$ は次のように入力列 w の長さに関して再帰的に定義することができる．

基礎：$\hat{\delta}(q,\varepsilon)=q$．すなわち，状態 q で何も入力を読まないとき，状態は q のままである．

再帰：列 w が xa の形だとする．すなわち，w の最後の記号が a でその前の部分が x だとする．たとえば，$w = 1101$ を $x = 110$ と $a = 1$ に分解する．このとき

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a) \tag{2.1}$$

とおく．式 (2.1) は一見複雑そうに見えるかも知れないがアイデアは単純である．$\hat{\delta}(q, w)$ を計算するには，まず 状態 q から始めて記号列 w の最後の記号を除いた部分 x によって到達する状態 $\hat{\delta}(q, x)$ を計算する．この状態を p とする．すなわち，$\hat{\delta}(q, x) = p$ とする．そのとき，求める $\hat{\delta}(q, w)$ は，w の最後の入力記号 a によって p から 遷移する先の状態に等しい．すなわち，$\hat{\delta}(q, w) = \delta(p, a)$ である．

例 2.4　次の言語を受理する DFA を求めよう．

$$L = \{w \mid w \text{ は偶数個の 0 と偶数個の 1 を含む }\}$$

容易に想像されるように，その DFA の状態では，それまでに読み込んだ 0 の個数と 1 の個数が偶数か奇数かを共に記憶する．すなわちそれらの状態は，それまでに読んだ 0 の個数が偶数か奇数かを記憶し，さらに 1 の個数が偶数か奇数かも記憶する．したがって，全部で次の四つの状態が必要である．

q_0: それまでに読んだ 0 の個数と 1 の個数は共に偶数である．

q_1: それまでに読んだ 0 の個数は偶数で，1 の個数は奇数である．

q_2: それまでに読んだ 1 の個数は偶数で，0 の個数は奇数である．

q_3: それまでに読んだ 0 の個数と 1 の個数は共に奇数である．

ここで，状態 q_0 は開始状態で，かつ唯一の受理状態である．それが開始状態である理由は，入力を読む前にはそれまでに読んだ 0 と 1 の個数は共に 0 で，0 は偶数だからである．それが唯一の受理状態である理由は，q_0 の条件そのものが，入力列が言語 L に属するための条件だからである．

言語 L に対する DFA は，次によって与えられる．

$$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

ここで，遷移関数 δ は図 2.6 の遷移図で示す．入力として 0 を読むと，状態は図の水平の点線を横切ることに注意せよ．よって，偶数個の 0 を読んだときの状態はこの線の上側の q_0 か q_1 であり，奇数個の 0 を読んだときの状態はこの

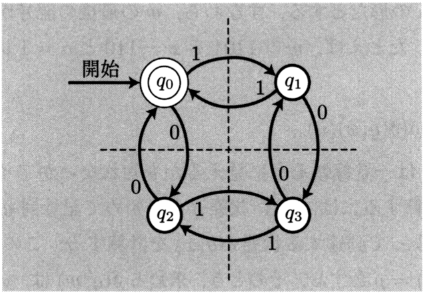


図 2.6　例 2.4 の DFA に対する遷移図

線の下側の q_2 か q_3 である．同様に，入力として 1 を読むと，状態は垂直の点線を横切る．したがって，偶数個の 1 を読んだときの状態はこの線の左側の q_0 か q_2 であり，奇数個の 1 を読んだときの状態はこの線の右側の q_1 か q_3 である．こうした観察は，これら四つの状態が上に述べた条件と対応していることを説明しているが，これらのことは例 1.23 で述べた同時帰納法により，厳密に示すことができる．

このDFA を遷移表を用いて示すこともできる．図 2.7 にその表を示す．ただし，この図を用いて単に DFA の構成を示すだけでなく，遷移関数 δ から $\hat{\delta}$ を作る方法についても説明することにする．今，入力が 110101 だとする．この列は偶数個の 0 と偶数個の 1 を含むから，これは上の言語 L に属する．ところで，q_0 は唯一の受理状態でもあるから，$\hat{\delta}(q_0, 110101) = q_0$ となるはずである．この主張を確認してみよう．

それを確かめるには，110101 のすべての接頭語（prefix）[3] w について最も

	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

図 2.7　例 2.4 の DFA のための遷移表

[3] ＜**訳注：**語 x, y, z の間に $xy = z$ の関係があるとき，x を z の接頭語という．＞

標準的記法と局所変数

この節を読んで，たとえば遷移関数はいつも δ という名前をつけなければならないとか，DFA の名前は常に A でなければならないというように，我々が用いた習慣的な記法を常に使わなければならないと考えた読者がいるかも知れない．我々がほとんどの例で，同じ種類のものには同じ変数名を使ったのは，たとえばプログラムの中で i という変数はふつう整数型の変数に使われる習慣があるのと同じように，そうすることによって読者がその変数の型を間違えないようにするためである．オートマトンの構成要素やその他のものの場合も他の場合と同様に，どういう名前をつけなければならないというきまりがある訳ではない．したがって，たとえば DFA を M と呼んでもよいし，その遷移関数を T と呼んでも何ら差し支えない．

また，同じ変数が別の文脈で別のものを表すことにも驚かないでほしい．たとえば，例 2.1 と 2.4 で異なるオートマトンの遷移関数が共に δ と呼ばれていた．ここで二つの遷移関数は共にそれぞれの例の中だけで通用する局所変数である．実際，これらの二つの遷移関数は別々のものであって，この両者の間に特に関係がある訳ではない．

短い ε から長いものへ順番に $\hat{\delta}(q_0, w)$ を計算する必要がある．この計算結果は次の通りである．

- $\hat{\delta}(q_0, \varepsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \varepsilon), 1) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$. ▨

2.2.5 DFA の言語

ここで DFA $A = (Q, \Sigma, \delta, q_0, F)$ の**言語** (language) を定義しよう．この言語は $L(A)$ で表され，その定義は

$$L(A) = \{\ w \mid \hat{\delta}(q_0, w) \text{ は } F \text{ に属する}\ \}$$

である．すなわち，A の言語は，開始状態 q_0 を受理状態の一つに導くような入

力列 w の全体である．L がある DFA A の言語 $L(A)$ と一致するとき，L を**正則言語**（regular language）と呼ぶ．

例 2.5 先に述べた通り，A が例 2.1 の DFA のとき，その言語 $L(A)$ は 0 と 1 から成る列で 01 を部分列として含むものの全体である．一方，A が例 2.4 の DFA のときは，その言語 $L(A)$ は 0 と 1 から成る列で 0 の個数と 1 の個数が共に偶数であるものの全体である． ▨

2.2.6 2.2 節の演習問題

♣ **問 2.2.1** ビー玉を転がして遊ぶ図 2.8 のような玩具について考える．図の A または B からビー玉を落すと，三つのレバー x_1, x_2, x_3 のところで左か右の道を進み，最後に出口 C または D から出る．ただし， ビー玉がレバーを通過すると，その後でレバーが反対方向に倒れ，次に来るビー玉は前と反対の方向に進むものとする．

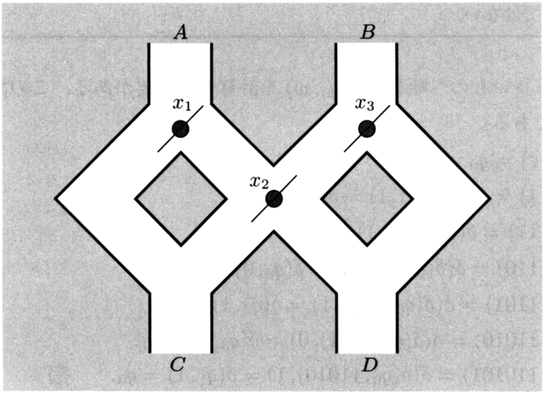


図 2.8 ビー玉転がしの玩具

*a) この玩具を有限オートマトンで表せ．ただし， ビー玉を A から落すことを入力 A で表し，ビー玉を B から落すことを入力 B で表す．また，ビー玉が出口 D から出ることを，受理状態と見なす．

!b) 上のオートマトンの言語はどんな集合か，ことばで説明せよ．

c) もし，レバーがビー玉の通った後ではなく，通る前に方向を変えるとすると，上の (a) と (b) の答はどう変わるか．

♣*! **問 2.2.2** 我々は $\hat{\delta}$ を定義する際，入力記号列を最後の記号とその前の部分に分割した（式 (2.1) を参照）．しかし，$\hat{\delta}$ は遷移図上の経路のラベルを繋いでできる列にその経路の行先を対応させる関数だから，$\hat{\delta}$ を定義する際，入力列をどう分解しても結果は変わらないはずである．任意の状態 q と記号列 x, y について $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ が成り立つことを示せ．**ヒント**：y に関する帰納法による．

♣ ! **問 2.2.3** 任意の状態 q と入力記号列 x と入力記号 a について $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$ であることを示せ．**ヒント**：演習問題 2.2.2 を利用せよ．

♣ **問 2.2.4** アルファベット {0,1} 上の次の言語を受理する DFA をそれぞれ求めよ．

*a) 00 で終わる列の全体．

b) 0 が（最後とは限らず，どこかに）3 個続いて現れる列の全体．

c) 011 を部分列として含む列の全体．

♣ ! **問 2.2.5** アルファベット {0,1} 上の次の言語を受理する DFA をそれぞれ求めよ．

a) 長さ 5 の部分列には常に少なくとも 2 個の 0 が含まれる列の全体．

b) 右端から 10 番目の記号が 1 である列の全体．

c) 01 で始まるか，または 01 で終わる列の全体．

d) その列の中に現れる 0 の個数が 5 で割り切れ，1 の個数は 3 で割り切れるような列の全体．

♣!! **問 2.2.6** アルファベット $\{0, 1\}$ 上の次の言語を受理する DFA をそれぞれ求めよ．

*a) 1 で始まり，2 進整数として見たとき 5 の倍数である列の全体．たとえば，$101, 1010, 1111$ はこの言語に属し，$0, 100, 111$ は属さない．

b) 左右逆にして 2 進整数として見たとき，5 で割り切れる列の全体．たとえば，$0, 10011, 1001100, 0101$ はこの言語に属す．

♣ **問 2.2.7** DFA A のある状態 q がすべての入力記号 a に対して $\delta(q, a) = q$ を満たすとする．このとき，すべての入力列 w に対して $\hat{\delta}(q, w) = q$ が成り立つことを w の長さに関する帰納法で示せ．

♣ **問 2.2.8** DFA A のある入力記号 a がすべての状態 q に対して $\delta(q, a) = q$ を満たすとする．

a) すべての $n \geq 0$ について $\hat{\delta}(q, a^n) = q$ が成り立つことを，n に関する帰納法により示せ．ただし，a^n は n 個の a から成る列を表す．

b) $\{a\}^* \subseteq L(A)$ かまたは $\{a\}^* \cap L(A) = \varnothing$ が成り立つことを示せ．

♣*! **問 2.2.9** DFA $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ において，すべての入力記号 a に対して $\delta(q_0, a) = \delta(q_f, a)$ が成り立つとする．

a) すべての $w \neq \varepsilon$ に対して $\hat{\delta}(q_0, w) = \hat{\delta}(q_f, w)$ が成り立つことを示せ．

b)　x が言語 $L(A)$ に属する空でない列とすると，すべての $k>0$ に対して x^k（x を k 回繰り返してできる列）も $L(A)$ に属することを示せ．

♣*! **問 2.2.10**　次の遷移表で表される DFA について考える．

	0	1
$\to A$	A	B
$*B$	B	A

この DFA で受理される言語をことばで説明し，それが正しいことを入力列の長さに関する帰納法で示せ．**ヒント：**各状態について，どんな入力を読むとその状態に到達するかを予想し，その予想を同時帰納法で証明せよ．

♣ ! **問 2.2.11**　前問と同じことを次の遷移表について考えよ．

	0	1
$\to *A$	B	A
$*B$	C	A
C	C	C

2.3　非決定性有限オートマトン

「非決定性」有限オートマトン（nondeterministic finite automaton，NFA と略す）とは，同時にいくつかの状態に到達することが可能なオートマトンである．非決定性とは，与えられた入力について何らかの「推測をする」能力といわれることがある．たとえば，オートマトンを使って長いテキストファイル中の文字列（たとえば，いくつかのキーワード）を検索する場合に，今そのうちのある文字列の先頭にいると推測して，その列が本当に現れるかどうかを，いくつかの状態を使って 1 文字ずつチェックすることを考えるとわかりやすい．この種の応用例について後の 2.4 節で述べる．

応用について調べる前に，非決定性有限オートマトンを定義し，それらは DFA によって受理される言語だけを受理することを示そう．つまり，NFA も DFA と同じく，正則言語を，そしてそれのみを受理する．しかし，それにも拘わらず NFA を考える理由は，NFA の方が DFA よりも簡潔でしかも作るのが容易である場合が多いためである．さらに，NFA を常に DFA に変換できるとはいうものの，後者は NFA よりも状態数が指数関数的に増加することがある．ただし，幸いなことにそのような場合は稀である．

2.3.1 非決定性有限オートマトンの直観的説明

DFA と同様に NFA も，状態の有限集合，入力記号の有限集合，一つの開始状態といくつかの受理状態を持つ．また，遷移関数を持ち，それをふつう δ で表す．DFA と NFA の違いは関数 δ の型の相違である．NFA の場合，δ は状態と入力記号の組に対してある値を返すが，その値が，DFA の場合のようにただ一つの状態でなく，(0 個か，1 個か，あるいはもっとたくさんの) 状態の集合である．以下で，まず NFA の例を示し，その後で正確な定義を述べる．

例 2.6 図 2.9 は 0 と 1 の列で，01 で終わるもののを受理し，それ以外の列を受理しない NFA の図である．状態 q_0 は開始状態であり，入力列の最後の 01 より前の部分を読んでいると推測される間，このオートマトンは q_0 の状態にとどまっていることができる．状態 q_0 から q_0 自身に向かう遷移が入力 0 に対しても 1 に対しても可能であるのはそのことを意味する．

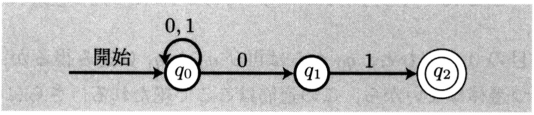


図 2.9 末尾が 01 の列を受理する NFA

しかし，入力記号が 0 のとき，この NFA はそれが末尾の 01 の 0 かも知れないと想像することもできる．状態 q_0 から q_1 へ向かうラベル 0 を持つ辺はこの想像に対応している．つまり，この NFA は q_0 で入力 0 を読んだとき，q_1 へ向かうというオプションと q_0 にとどまるというオプションを共に用意しているのである．状態 q_1 では次の記号が 1 かどうかを調べ，もしそうなら状態 q_2 に進みその入力は受理される．

状態 q_1 からラベル 0 のついた辺は一つも出ていないことに注意せよ．また，状態 q_2 から出る辺はまったくない．このことは，この NFA の上をここまで進んできた道筋はこの先どこにも行き場がないこと，つまり前途が絶たれたことを意味する．ただし，同じ入力でこれ以外の経路を進んだものがあればそれはまだ生きている．DFA の場合は，ある状態から一つの入力記号を読んで到達できる状態はちょうど一つだったが，NFA の場合はそのような制限はない．図 2.9 の場合に見られるように，一つの状態から同じラベルのついた辺が 0 本，1

本，2本あるものなど様々の可能性がある．

図2.10は，NFAがどのように入力を処理するかを図示している．この図は図2.9のオートマトンが入力列として00101を読み込んだとき何が起きるかを表す．開始状態 q_0 を出発して最初の記号0を読むとこのNFAは状態 q_0 か q_1 に進み得る．図の最も左にある二つの矢印がその二つの可能性を示している．

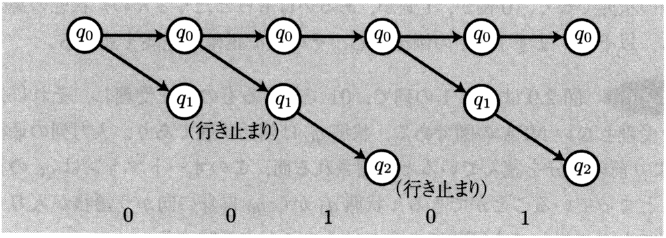


図2.10　入力列00101に対するあるNFAの辿る経路

次に2番目の0を読むと，q_0 からは再び q_0 か q_1 に進み得るが，q_1 からラベル0を持つ遷移はないから，この道筋はここで絶たれる．さらに3番目の入力記号1を読むと，q_0 からは q_0 にのみ進み得るが，q_1 からは q_2 にのみ進み得る．したがって，入力列として001を読んだ段階では，このNFAの可能な状態は q_0 と q_2 である．しかるに q_2 は受理状態だから，入力列001はこのNFAによって受理される．

しかし，入力はまだ続く．4番目の入力記号0で q_2 から先には進めないが，q_0 からは q_0 と q_1 に進める．そして最後の入力記号1によって，q_0 からは q_0 へ，q_1 からは q_2 に進める．よってこのときも受理状態にいることが可能だから，00101はこのNFAで受理される．▨

2.3.2 非決定性有限オートマトンの定義

以下で，非決定性有限オートマトン（NFA）に関する諸定義を，DFAの場合との違いを指摘しながら述べることにする．NFAはDFAの場合と同じように次の形で与えられる．

$$A = (Q, \Sigma, \delta, q_0, F)$$

ただし，

(1) Q は**状態**（state）の有限集合.

(2) Σ は**入力記号**（input symbols）の有限集合.

(3) q_0 は Q の要素で，**開始状態**（start state）と呼ばれる.

(4) F は Q の部分集合で，**最終状態**（final state）または**受理状態**（accepting state）の集合と呼ばれる.

(5) δ は，状態（Q の要素）と入力記号（Σ の要素）に対して状態の集合を返す**遷移関数**（transition function）である．NFA と DFA の唯一の違いはこの δ の返す値の型（DFA の場合は単一の状態なのに対して，NFA の場合は状態の集合）であることに注意せよ.

例 2.7 図 2.9 の NFA は正式には

$$(\{q_0, q_1, q_2\},\ \{0,1\},\ \delta,\ q_0,\ \{q_2\})$$

で表される．ここで，遷移関数 δ は図 2.11 の遷移表で示す. ▨

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

図 2.11 末尾が 01 である列の全体を受理する NFA の遷移表

DFA の場合と同様に，NFA の場合も遷移表によって遷移関数を指定できる．ただし，NFA の遷移表の場合は，値として表に書き込むものは状態の集合である（ただし，ただ一つの状態だけから成る単元集合でもよい）点が DFA の場合とは異なる．特に NFA では，ある状態からある入力に対する遷移が一つもないこともあり，その場合は遷移表の該当する欄に空集合 $\emptyset$ を記入する.

2.3.3 遷移関数の拡張

DFA の場合と同じように，NFA の遷移関数 δ を $\hat{\delta}$ に拡張する必要がある．ただし，$\hat{\delta}$ は，状態 q と入力記号列 w に対して，その NFA が状態 q から出発して入力列 w を読んだとき到達することのできる状態全体の集合を値として返す関数である．その考え方はすでに図 2.10 で示唆されている．この図の最初の列が q だけのとき，状態 q から入力 w を読んだあとに到達可能な状態が縦に並んで現れるが，それらの状態全体の集合が $\hat{\delta}(q, w)$ である．たとえば，図 2.10 は $\hat{\delta}(q_0, 001) = \{q_0, q_2\}$ であることを示している．厳密にいうと，NFA の $\hat{\delta}$ は

遷移関数 δ から以下のように定義される.

基礎：$\hat{\delta}(q,\varepsilon)=\{q\}$. すなわち, 状態 q で何も入力を読まないとき, 到達可能な状態は q のみである.

再帰：列 w が xa だとする. ただし a は w の最後の入力記号で, x は w のそれ以外の部分を表す. さらに $\hat{\delta}(q,x)=\{p_1,p_2,\ldots,p_k\}$, かつ

$$\bigcup_{i=1}^{k}\delta(p_i,a)=\{r_1,r_2,\ldots,r_m\}$$

とする. このとき, $\hat{\delta}(q,w)=\{r_1,r_2,\ldots,r_m\}$ である. つまり, $\hat{\delta}(q,w)$ を求めるには, まず $\hat{\delta}(q,x)$ を計算し, そこに含まれる状態から入力記号 a で遷移する先をすべて集めればよい.

例 2.8　図 2.9 の NFA に入力列 00101 を与えたときに実行されるプロセスを見るため, $\hat{\delta}$ を計算してみよう.

(1)　$\hat{\delta}(q_0,\varepsilon)=\{q_0\}$.

(2)　$\hat{\delta}(q_0,0)=\delta(q_0,0)=\{q_0,q_1\}$.

(3)　$\hat{\delta}(q_0,00)=\delta(q_0,0)\cup\delta(q_1,0)=\{q_0,q_1\}\cup\varnothing=\{q_0,q_1\}$.

(4)　$\hat{\delta}(q_0,001)=\delta(q_0,1)\cup\delta(q_1,1)=\{q_0\}\cup\{q_2\}=\{q_0,q_2\}$.

(5)　$\hat{\delta}(q_0,0010)=\delta(q_0,0)\cup\delta(q_2,0)=\{q_0,q_1\}\cup\varnothing=\{q_0,q_1\}$.

(6)　$\hat{\delta}(q_0,00101)=\delta(q_0,1)\cup\delta(q_1,1)=\{q_0\}\cup\{q_2\}=\{q_0,q_2\}$.

(1) は $\hat{\delta}$ の再帰的定義の基礎である. (2) は, $\hat{\delta}(q_0,0)$ を上の再帰的ステップにしたがって求めている. ただし, ここで $0=\varepsilon 0$ と分解し, (1) の結果 $\hat{\delta}(q_0,\varepsilon)=\{q_0\}$ を使っている. (3) は, (2) で得られた二つの状態 q_0,q_1 から新しい入力記号 0 でそれぞれ遷移し得る状態の集合 $\delta(q_0,0)=\{q_0,q_1\}$ と $\delta(q_1,0)=\varnothing$ の和集合を求めている. (4) も同様に, (3) で得られた二つの状態から新しい入力記号 1 でそれぞれ遷移し得る状態の集合 $\delta(q_0,1)=\{q_0\}$ と $\delta(q_1,1)=\{q_2\}$ の和集合を求めている. (5),(6) はそれぞれ (3),(4) と同様である. ▨

2.3.4 NFA の 言 語

前にも触れたように, NFA は入力記号列 w を読みながら許された状態遷移に従い開始状態から受理状態の一つに到達できるとき, その入力列 w を受理する. 同じ入力列 w に対して別の経路を進むと受理状態に到達できない（つまり, 受理状態でない状態に到達したり, あるいはどんな状態にも到達できず経路が

絶たれる）としても，そのことを気にする必要はない．数学的に正確にいうと，NFA $A = (Q, \Sigma, \delta, q_0, F)$ に対して

$$L(A) = \{\ w \mid \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$

と定義する．すなわち，$\hat{\delta}(q_0, w)$ が少なくとも一つ受理状態を含むような入力記号列 w $(\in \Sigma^*)$ の全体を $L(A)$ とおく．

例 2.9　例として，図 2.9 の NFA A が $L = \{\ w \mid w$ は 01 で終わる $\}$ を受理することを上の定義にしたがって厳密に証明してみよう．そのため，A の三つの状態の性質を述べた次の命題を同時帰納法により証明する．

(1)　各 w に対して，$\hat{\delta}(q_0, w)$ は q_0 を含む．

(2)　$\hat{\delta}(q_0, w)$ が q_1 を含むためには，w が 0 で終わることが必要十分条件である．

(3)　$\hat{\delta}(q_0, w)$ が q_2 を含むためには，w が 01 で終わることが必要十分条件である．

これらの命題を示すには，A が各状態にどのようにして到達するか（すなわち，直前に読み込んだ入力記号と，その記号を読む前に A はどの状態にいたか）について考える必要がある．このオートマトンでは，q_2 が唯一の受理状態だから，その言語は，$\hat{\delta}(q_0, w)$ が q_2 を含むような w の全体である．よって，上の三つの命題（特に命題 (3)）が示せれば，この NFA が受理する言語は 01 で終わる列の全体であることがわかる．これらの命題の証明は，w の長さ $|w|$（0 以上の整数）に関する帰納法による．

基礎：$|w| = 0$ のとき，$w = \varepsilon$ である．命題 (1) はこのとき，$\hat{\delta}(q_0, \varepsilon)$ が q_0 を含むと主張しているが，これは $\hat{\delta}$ の定義の基礎部分より明らかである．命題 (2) については，ε の末尾が 0 でないことと，$\hat{\delta}(q_0, \varepsilon)$ が q_1 を含まないこと（これも $\hat{\delta}$ の定義の基礎部分による）より，命題 (2) の二つの条件は共に偽である．よってこの命題自体は真である．$w = \varepsilon$ に対する命題 (3) の証明は (2) の場合と同様である．

帰納：$w = xa$，ただし a は入力記号 0 または 1 で x は記号列とする．ここで，x について三つの命題は成り立っていると仮定して，w についてこれらの命題が成り立つことを示す．すなわち，$|w| = n + 1$ かつ $|x| = n$ で，n について帰納法の仮定が成り立つとして，$n + 1$ の場合を示す．

(1) 帰納法の仮定から $\hat{\delta}(q_0, x)$ は q_0 を含む．一方，q_0 から 0 と 1 のラベルの付いた自分自身への遷移があるから，したがって $\hat{\delta}(q_0, w)$ も q_0 を含む．ゆえに，w についても (1) は成り立つ．

(2) （十分性）w が 0 で終わるとき，すなわち $a = 0$ のとき，(1) が x について成り立つ（帰納法の仮定）から $\hat{\delta}(q_0, x)$ は q_0 を含み，q_0 から入力 0 で q_1 に向かう遷移がある．したがって，$\hat{\delta}(q_0, w)$ は q_1 を含む．
（必要性）$\hat{\delta}(q_0, w)$ が q_1 を含む場合，図 2.9 の遷移図を見ると，状態 q_1 に到達するのは，入力列 w が $x0$ の形の場合に限ることは明らかである．

(3) （十分性）w が 01 で終わるとき，$w = xa$ とすると $a = 1$ かつ x は 0 で終わる．そこで，x について命題 (2) が成り立つことから，$\hat{\delta}(q_0, x)$ は q_1 を含む．さらに，q_1 から入力 1 で q_2 に向かう遷移があることから，結局，$\hat{\delta}(q_0, w)$ は q_2 を含むことがわかる．
（必要性）$\hat{\delta}(q_0, w)$ が q_2 を含むとする．図 2.9 の遷移図を見ると，状態 q_2 に到達するのは，w が $x1$ の形で $\hat{\delta}(q_0, x)$ が q_1 を含む場合に限ることがわかる．ここで，x について命題 (2) が成り立つこと（帰納法の仮定）から，x は 0 で終わる．したがって，w は 01 で終わることがわかる．以上で命題 (3) が示された．▨

2.3.5 決定性と非決定性の有限オートマトンの等価性

たとえば，例 2.6 の言語（01 で終わる列の全体）の場合がそうであったように，多くの言語について，それを受理する NFA を作るのは DFA を作るより容易である．しかし，驚くべきことに，NFA の受理集合は DFA でも受理することができる．さらに，DFA で記述する場合に必要な状態遷移の数は NFA の場合より大抵多いが，状態数については多くの場合，DFA と NFA の間で大差はない．しかし，最悪の場合についていうと，同じ言語を受理するのに，DFA では少なくとも 2^n 個の状態を必要とするのに，NFA では n 個の状態で十分という場合がある．

NFA にできることは DFA にもできることを示すには，サブセット構成（subset construction）と呼ばれる重要な構成法を用いる．その名前の由来は，この構成法では NFA の状態の集合の部分集合の全体に注目して，その NFA と同じ言語を受理する DFA を構成することによる．この方法は，オートマトンが一つ

与えられたとき，その性質について特別な情報のないまま，その状態集合と遷移関数のみに基づいて新しいオートマトンを構成する方法の一つとして重要なものである．

NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ が与えられたとする.「サブセット構成」の目標は，$L(N) = L(D)$ を満たす DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ を構成することである．ここで，二つのオートマトンの入力アルファベット（入力記号の全体）は等しく，かつ，D の開始状態は N の開始状態 q_0 を唯一の要素とする集合 $\{q_0\}$ であることに注意せよ．D の他の要素は以下のように構成される．

- Q_D は Q_N の部分集合全体から成る集合 (すなわち，Q_N の**冪集合**) である．したがって，もし Q_N が n 個の状態から成るとき，Q_D は 2^n 個の状態から成る．ただし，その 2^n 個の状態のすべてが Q_D の開始状態から到達できるとは限らない．到達できない状態は「捨てる」ことができるので，実際には，D の状態数は 2^n よりずっと少ないかも知れない．
- F_D は Q_N の部分集合 S で，$S \cap F_N \neq \varnothing$ を満たす S から成る集合である．すなわち，D の状態 S $(\subseteq Q_N)$ が受理集合であるのは，S が N の受理状態を少なくとも一つ含むとき，かつそのときに限る．
- 各 S $(\subseteq Q_N)$ と各入力記号 a $(\in \Sigma)$ に対して，
$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$$
とおく．すなわち，$\delta_D(S, a)$ を求めるには，集合 S に属するすべての $p \in S$ について N が入力 a で p からどの状態に遷移できるかを調べ，それらの集合和をとればよい．

例 2.10 N を，末尾が 01 の列全体を受理する図 2.9 のオートマトンとする．N の状態の集合は $(\{q_0, q_1, q_2\}$ であるから，これにサブセット構成を適用すると，この集合の部分集合のそれぞれに対応する $2^3 = 8$ 個の状態を持つ DFA ができる．図 2.12 はこれらの 8 個の状態に対する遷移表を示している．この表のいくつかの欄の詳しい計算法を述べる．

この遷移表は決定性有限オートマトンの遷移表であることに注意せよ．表の各欄には集合が記載されているが，それはいま構成している DFA の状態が集合だからである．注目すべきポイントを明確にするため，これらの状態に新しい呼び名，たとえば $\varnothing$ に A，$\{q_0\}$ に B，などを付けてみよう．そのように状態

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\varnothing$	$\{q_2\}$
$*\{q_2\}$	$\varnothing$	$\varnothing$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\varnothing$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

図 2.12 図 2.9 の遷移表にサブセット構成を適用した結果

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

図 2.13 図 2.12 の状態の名前を付け変えた結果

の名前を付け変えてできる図 2.13 の遷移表は図 2.12 と同じオートマトンを表しているが，これをみると確かに表の各欄は単一の状態が記載された DFA の遷移表であることがわかる．

図 2.13 の 8 個の状態の中で，状態 B から出発して到達できる状態は B, E, F のみである．残りの状態は開始状態から到達できないから存在しないも同然である．上のように状態集合のすべての部分集合について遷移表の欄を埋めていくと指数関数的な時間が掛かるが，その代わりに，次に述べる「必要なものをその都度取り込む」方針にしたがって遷移表を構成していくと時間が節約できる可能性がある．

基礎： N の開始状態のみから成る単元集合は (その状態自身から) 確かに到達可能である．

帰納：D の状態 S $(\subseteq Q_N)$ が（開始状態から）到達可能[4]なとき，各入力記号 a に対して前々ページの方法で求めた状態 $\delta_D(S, a)$ もやはり到達可能である．

図 2.12 の例の場合，$\{q_0\}$ は DFA D の（開始状態から到達可能な）状態である．次に，$\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ かつ $\delta_D(\{q_0\}, 1) = \{q_0\}$ であることが，図 2.9 の遷移図から直ちに見てとれる（実際，入力 0 で $\{q_0\}$ から $\{q_0\}$ と $\{q_1\}$ に向かう辺があることと，入力 1 で $\{q_0\}$ から $\{q_0\}$ へ向かう辺だけがあることに注意せよ）．このようにして，DFA に対する遷移表の一つの行（図 2.12 の 2 行目）が得られる．

このようにして得た二つの集合のうち，$\{q_0\}$ は今新しく得られたものではない．しかし，もう一方の $\{q_0, q_1\}$ は今回始めて登場した集合であるから，この集合に対する遷移表をさらに求める必要がある．定義から $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ かつ $\delta_D(\{q_0, q_1\},\ 1) = \{q_0, q_2\}$ となる．たとえば後者の場合を詳しくみると

$$\delta_D(\{q_0, q_1\},\ 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$$

こうして図 2.12 の 5 行目が得られ，DFA D の新しい状態として今度は $\{q_0, q_2\}$ が登場した．同様な計算をすると

$$\delta_D(\{q_0, q_2\},\ 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$$

$$\delta_D(\{q_0, q_2\},\ 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \varnothing = \{q_0\}$$

このようにして図 2.12 の 6 行目が得られるが，それによって新たに登場する状態は一つもない．

以上でサブセット構成は終了した．つまり，到達可能な状態とそれに対する遷移はすべて出尽くした．その結果できた DFA の遷移図を図 2.14 に示す．状態はたった三つ（たまたま，元の図 2.9 の NFA の状態数と同数）しかないことに注意せよ．しかし，元の NFA が 4 個の遷移を持つのに対して，図 2.14 の DFA は 6 個の遷移を持っている． ▨

上の例から直観的にサブセット構成が正しく働くことが理解されたことと思うが，その正しさを下の定理で正確に証明してみよう．証明のアイデアは次の通りである．まず，入力記号列 w を読んだときの DFA の状態は，元の NFA が

---

4) ＜**訳注**：オートマトンの開始状態から到達可能な状態を以後単に到達可能（accessible）な状態という．＞

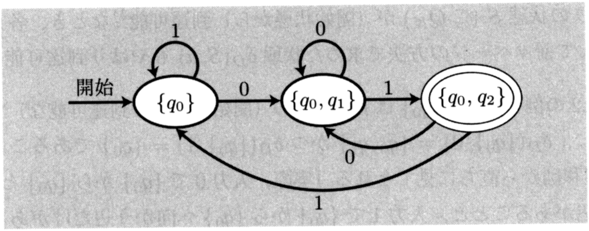


図 2.14 図 2.9 の NFA から構成した DFA

同じ入力列 w を読んだとき到達し得る状態全体の集合であることに注意する．さらに，NFA の受理言語の定義と DFA の受理状態の定義から，DFA と NFA がちょうど同じ列を受理することが結論できる．

定理 2.11 NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ からサブセット構成によって DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ が作られたとき，$L(D) = L(N)$ が成り立つ．

証明 はじめに，$|w|$ に関する帰納法により

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

を示す．ここで注意すべきことは，上式の両辺の $\hat{\delta}$ 関数の値は共に Q_N の状態の集合であること，しかし，左辺の $\hat{\delta}_D$ の場合，それは Q_D（つまり，Q_N の冪集合）に属する状態の一つと見なされるのに対して，右辺の $\hat{\delta}_N$ は単に Q_N の部分集合としての役割のみを演じていることである．

基礎：$|w| = 0$，すなわち $w = \varepsilon$ とする．このとき，DFA と NFA に対する $\hat{\delta}$ の定義から，$\hat{\delta}_D(\{q_0\}, \varepsilon)$ と $\hat{\delta}_N(q_0, \varepsilon)$ は共に $\{q_0\}$ である．

帰納：w の長さが $n+1$ で，長さ n の列については定理が成り立っていると仮定する．w を $w = xa$ と分解する．ただし，a は w の最後の記号である．このとき，帰納法の仮定から，$\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$ である．N の状態から成るこの集合を $\{p_1, p_2, \ldots, p_k\}$ とする．

NFA に対する $\hat{\delta}$ の定義によれば，

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

他方，サブセット構成によれば，

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

ここで式 (2.3) と $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ の事実を用いて，DFA に対する $\hat{\delta}$

の定義の式を書き換えると，

$$
\begin{aligned}
\hat{\delta}_D(\{q_0\}, w) &= \delta_D(\hat{\delta}_D(\{q_0\}, x), a) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) \\
&= \bigcup_{i=1}^{k} \delta_N(p_i, a) \qquad (2.4)
\end{aligned}
$$

したがって，式 (2.2) と (2.4) より $\hat{\delta}_N(q_0, w) = \hat{\delta}_D(\{q_0\}, w)$．ここで，$D$ も N も列 w を受理するのは，$\hat{\delta}_N(q_0, w)$ あるいは $\hat{\delta}_D(\{q_0\}, w)$ が F_N の要素を含むとき，かつそのときに限るから，結局 $L(D) = L(N)$ が導かれる． ▨

定理 2.12 言語 L がある DFA で受理されるための必要十分条件は L がある NFA で受理されることである．

証明 （十分性）サブセット構成と前定理による．

（必要性）この方向はやさしい．というのは，DFA をそれと等価な（すなわち，同じ言語を受理する）NFA に変換すればよいからである．直観的にいうと，DFA の遷移図が与えられたとき，それを（たまたま，どの状況においても可能な遷移がただ一つある）NFA の遷移図と見ることができる．より厳密にいうと，$D = (Q, \Sigma, \delta, q_0, F)$ が DFA のとき，それと等価な NFA $N = (Q, \Sigma, \delta_N, q_0, F)$ を定義するには，δ_N を

$\delta_D(q, a) = p$ の場合，$\delta_N(q, a) = \{p\}$

と定義すればよい．このとき，$|w|$ に関する帰納法により

$\hat{\delta}_D(q, w) = p$ の場合，$\hat{\delta}_N(q, w) = \{p\}$

であることが容易に導かれる．その証明は読者に委ねる．このことから，w が D によって受理されるとき，かつそのときに限り，w は N によって受理されること，つまり $L(D) = L(N)$ であることがわかる． ▨

2.3.6 サブセット構成で状態数が増える場合

例 2.10 の場合，NFA と等価な DFA の状態数は前より増えなかった．前にも述べたように，現実にはこのように NFA をもとに構成した DFA の状態数がもとと余り違わないことが多い．しかし，DFA の状態数が NFA のそれに比べて指数関数的に増大すること，すなわち n 個の状態を持つ NFA からサブセット構成によって作った DFA の 2^n 個の状態がすべて到達可能なことは現実にあり得る．次の例はその上限に達してはいないが，$n+1$ 個の状態を持つ NFA と等価な最小の DFA が 2^n 個の状態数を持つ場合である．

例 2.13　図 2.15 の NFA N について考える．このとき $L(N)$ は，0 と 1 の列で，最後から n 番目の記号が 1 であるもの全体の集合である．この言語を受理する DFA D は，直観的にいって，最後に読んだ n 個の記号を記憶しておく必要があり，そのために全部で 2^n 個以上の状態を必要とする．実際，もし D が 2^n より少ない個数の状態しか持たないとすると，次のような状況が起きる：長さ n の二つの異なる入力列 $a_1a_2\cdots a_n$ と $b_1b_2\cdots b_m$ によって D は（開始状態から）ある同じ状態 q に到達する．

この二つの列はどこかで異なるから $a_i \neq b_i$ と仮定する．さらに，対称性により $a_i = 1$, $b_i = 0$ としても一般性は失われない．ここで，もし $i = 1$ とすると，$a_1a_2\cdots a_n$ は（N で受理され，N と D は等価だから）D でも受理される．したがって q は受理状態でなければならない．一方，$b_1b_2\cdots b_n$ は D で受理されないはずだから，q は受理状態ではない．$i > 1$ の場合は，$a_1a_2\cdots a_n$ の後に $i-1$ 個の 0 を付けた長さ $n+i-1$ の入力列を D が読んだときに到達する状態を p とすると，$a_1a_2\cdots a_n00\cdots0$ は D で受理され，$b_1b_2\cdots b_n00\cdots0$ は受理されないから，p は D の受理状態であって同時に受理状態でないことになる．

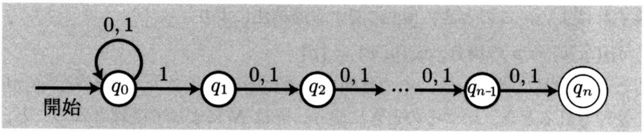


図 2.15　2^n 未満の状態を持つ等価な DFA が存在しない NFA

図 2.15 の NFA N の受理言語 $L(N)$ が確かに最後から n 番目の記号が 1 である 0 と 1 の列全体であることは以下のようにして確かめられる．N の開始状態 q_0 は，どんな入力列を読んでもここに留まることのできる状態である．ただし，次の入力が 1 のときは，これを最後から n 番目の記号だと予想して状態 q_1 に進んでもよい．状態 q_1 からは入力記号を一つ読むと q_2 に進み，次の入力記号で q_3 へ，と順に進んでいき，$n-1$ 個目の記号で受理状態 q_n に到達する．N の状態の性質を整理すると次のようになる．

(1)　N は，どんな入力列 w を読んだ後でも状態 q_0 に留まることができる．

(2)　N が入力列 w を読んで状態 q_i（ただし i は $1, 2, \cdots, n$ のいずれかとする）にいるのは，w の最後から i 番目の記号が 1（すなわち，w が適当な

入力記号 $a_1a_2\cdots a_{i-1}$ に対して $x1a_1a_2\cdots a_{i-1}$ の形）のとき，かつそのときに限る．

この証明は，$|w|$ に関する簡単な帰納法でできるが，ここでは省略する．我々の目標は，図 2.15 の NFA の受理言語が，最後から n 番目の記号が 1 である列全体であることだった．そのために上の命題 (2) で $i=n$ のときを考える．その場合 (2) は，N が状態 q_n に到達するとき，かつそのときに限り，入力列の右端から n 番目の記号が 1 である，と述べている．ところで，q_n は唯一の受理状態であるから，上の条件は N で受理される入力列の条件でもある． ▨

2.3.7 2.3 節の演習問題

♣ * **問 2.3.1** 次の NFA を DFA に変換せよ．

	0	1
$\rightarrow p$	$\{p,q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
$*s$	$\{s\}$	$\{s\}$

♣ **問 2.3.2** 次の NFA を DFA に変換せよ．

鳩の巣原理

例 2.13 で我々は**鳩の巣原理**の名で知られる重要な推論方法を用いた．それは平たくいうと，もし鳩の巣の数よりも多くの鳩がいて，彼らが巣に入るなら，少なくとも一つの巣に複数の鳩がいる，というものである．我々の例の場合,「鳩」に相当するのが長さ n の列で,「鳩の巣」に相当するのは状態である．上の例の場合，列の数が状態の数よりも少ないから，ある状態に二つ以上の列が割り当てられることになる．

鳩の巣原理はいつでも成り立つように思われるかも知れないが，鳩の巣が有限個の場合に限って成り立つ．したがって，有限状態のオートマトンにはこの原理が使えるが，別の種類の（無限個の状態を持つ）オートマトンには使えない．

鳩の巣原理において鳩の巣が有限個であることがなぜ本質的かは次のように説明できる．仮に無限の鳩の巣があり，それらを $1,2,\ldots$ としてみよう．また，鳩が巣より一つだけ多くいたとして，それらを $0,1,2,\ldots$ とする．しかし，このとき各 i について，鳩 i を巣 $i+1$ に入れれば，無限羽の鳩は無事に 1 羽ずつ巣に入り，2 羽の鳩が一つの巣を共有する必要はない．

遷移の欠けた DFA と死状態

我々は先に DFA の定義で，どんな状態でどんな入力記号を読んでもちょうど一つの状態への遷移がある，と述べた．しかし，DFA を設計する際，それ以上進んでも入力列が受理される可能性が全くない状態については，その状態からの遷移をしばしば省略する．たとえば，例 1.2 で述べた，ただ一つのキーワード `then` を認識するオートマトンを思い出してほしい．このオートマトンでは多くの遷移が省略されているため厳密にいうとこれは DFA ではない．

しかし，そのような不完全な DFA は NFA ではある．そして，これにサブセット構成を適用することにより，完全な DFA を得ることができる．そのようにしてできた DFA はほとんど元のものと同じだが，**死状態**（dead state）——この状態でどんな入力を読んでも自分自身に留まっている状態で受理状態でないもの——を含む．この死状態は，サブセット構成における $\varnothing$（図 1.2 の状態の空集合）に対応している．

一般に，一つの状態から同じ入力記号に対する遷移が二つ以上出ていないオートマトンがあるとき，新たに死状態を加えることによって，それを厳密な意味での DFA に作り変えることができる．そのため，上で述べた形の（一つの状態から同じ入力記号に対する遷移が二つ以上出ていない）オートマトンも DFA と呼ぶ場合がある．

	0	1
$\to p$	$\{q,s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q,r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\varnothing$	$\{p\}$

♣ **!問 2.3.3** 次の NFA を DFA に変換し，それが受理する言語をことばで説明せよ．

	0	1
$\to p$	$\{p,q\}$	$\{p\}$
q	$\{r,s\}$	$\{t\}$
r	$\{p,r\}$	$\{t\}$
$*s$	$\varnothing$	$\varnothing$
$*t$	$\varnothing$	$\varnothing$

♣ **!問 2.3.4** 次の各言語を受理する非決定性有限オートマトンを求めよ．その際，非決定性の利点をできるだけ活用せよ．

***a)** アルファベット $\{0,1,\ldots,9\}$ 上の列で，末尾の記号がそれより前にも現れ

るものの全体

b) アルファベット $\{0, 1, \ldots, 9\}$ 上の列で，末尾の記号がそれより前に現れないものの全体

c) 0 と 1 の列で，その中に二つの 0 が 4 の倍数の間隔をおいて現れているものの全体（0 は 4 の倍数であることに注意）

♣ **問 2.3.5** 定理 2.12 の必要性の証明の中で，$|w|$ に関する帰納法で $\hat{\delta}_D(q, w) = p$ のとき，$\hat{\delta}_N(q, w) = \{p\}$ を示す部分を省略した．その証明を補え．

♣ **! 問 2.3.6** 「遷移の欠けた DFA と死状態」の囲み記事の中で，N が NFA で，一つの状態から同じ入力記号に対する遷移が高々一つのとき（すなわち，$\delta(q, a)$ は 2 個以上の要素を持たないとき），N からサブセット構成によって作られる DFA D はほとんど元の N と同じで，ただし，死状態と，死状態に向かう遷移が加わる可能性があることを述べた．このことを証明せよ．

♣ **問 2.3.7** 例 2.13 で，NFA N が入力列 w を読んだ後で状態 q_i に到達するのは，w の末尾から i 番目の記号が 1 のとき，かつそのときに限ると述べた（ただし，$i = 1, 2, \ldots, n$）．このことを証明せよ．

2.4 応用：テキスト検索

前節で記号列の最後が 01 であるか否かという問題を考えたが，この節では，その種の議論がウェブ検索やテキストからの情報抽出等の実際的な問題にどのように応用されるかを見よう．

2.4.1 テキスト中の文字列を見つける

ウェブやその他のオンラインテキストが普及した今日，次のような仕事が頻繁に必要とされている： すべての文書の中から指定されたいくつかの語の一つ（またはすべて）を含むものを見付けよ．検索エンジンはこの種のプロセスの有名な例である．検索エンジンは**逆索引法**（inverted indexes）と呼ばれる特殊な技術を使っている．この方法では，ウェブに登場する（全部で 100,000,000 個もの）語のそれぞれに対して，その語が登場する場所のリストが記憶され，非常に大量の主記憶を持つコンピュータが頻度の高い語のリストを記憶し，大勢の人々が瞬時にその情報を検索できるようにしている．

逆索引法では有限オートマトンは使われずに，ウェブをコピーしたり，インデックス（索引項目）を整えるのに多くの時間を費している．一方この方法には不向きな問題で，オートマトンに基づく方法に適したものも数多くある．オー

トマトンを使う検索に適した応用の特徴を挙げると以下の通りである．

(1) 検索する対象がたびたび変化すること．たとえば

(a) ニュース解説者は毎日その日のオンラインニュースの記事の中に自分が関心を持つトピックスがあるかどうか検索する必要がある．たとえば，財政問題の解説者なら株式相場や会社名を検索するかも知れない．

(b) 「ショッピング・ロボット」は顧客が要求する商品の現在の価格を検索する必要があるかも知れない．このロボットは，最新のカタログの該当するページをまずウェブで検索し，次にそれらのページで個々の商品の価格を検索するかも知れない．

(2) 検索する文書がカタログ化できないこと．たとえば `Amazon.com` はその会社が販売するすべての本のすべてのページをゆっくり見るのには向いていない．むしろ，これらのページは要求に応じてその都度作られるのに適している．そして，それを使うには，まずあるトピックス（たとえば「有限オートマトン」）に関する本について検索し，次にその中でたとえばレビュー欄に「卓越した」という語があるものを探す，という使い方がされる．

2.4.2 テキスト検索のための非決定性有限オートマトン

キーワードと呼ばれるいくつかの語が与えられたとして，テキスト中にこれらの語が現れている箇所を見付けたいとする．このような場合に有効な方法として，非決定性有限オートマトンを作り，受理状態に到達したら，キーワードの一つが見付かったことを合図させることが考えられる．問題のテキストをこの NFA に1文字ずつ読み込ませて，テキスト中のキーワードを検出させるのである．そのための NFA は次のように構成すればよい．

(1) すべての入力記号に対して開始状態から開始状態自身に向かう遷移を付ける．ただし，ここでの入力記号は，テキストファイルを検索するのであれば，印刷可能なすべての ASCII 文字である．直観的にいうと，開始状態はまだキーワードを見ていないとの「予想」を表す（実際には，それらの語の一部をすでに見ていても構わない）．

(2) それぞれのキーワード $a_1a_2\cdots a_k$ に対して，k 個の状態，たとえば $q_1, q_2, \cdots, q_k$ を設け，開始状態で入力記号 a_1 を読んだら q_1 へ進む遷移，

q_1 で入力記号 a_2 を読んだら q_2 へ進む遷移，等々を加え，最後の q_k を受理状態としてそこに到達したらキーワード $a_1a_2\cdots a_k$ が見付かった旨を知らせる．

例 2.14 `web` と `ebay` という二つの語が現れるか否かを認識する NFA を設計したいとする．そのために上の考え方に基づいて設計した NFA の遷移図を図 2.16 に示す．状態 1 は開始状態である．Σ ですべての印刷可能な ASCII 文字の集合を表す．状態 2 から 4 までは `web` を認識するために使われ，状態 5 から 8 までは `ebay` の認識のために使われる．

もちろん，この NFA はプログラムではない．この NFA を実装するには次の二つの選択肢がある．

(1) この NFA の動作を模倣するプログラムを書く．そのために，各入力記号を読み込んだあとにこの NFA が到達できる状態の集合を計算する必要がある．このプログラムの考え方については図 2.10 を参考にせよ．

(2) この NFA をサブセット構成の方法により DFA に変換する．そして，DFA を直接模倣するプログラムを書く．

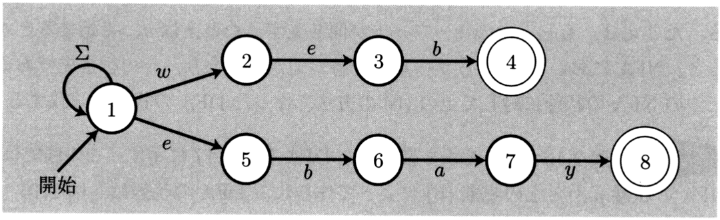


図 2.16 `web` と `ebay` を検索する NFA

ある種のテキスト処理プログラム，たとえば UNIX の `grep` コマンドの進んだ形（`egrep` と `fgrep`）では，実際，上に述べた二つのアプローチを混ぜて用いている．しかし，我々の目的にとって，DFA に変換する方が容易で，かつこの場合状態数も増加しないことが保証されている．▨

2.4.3 複数のキーワードを認識する DFA

サブセット構成は，どんな NFA に対しても適用することができる．しかし，キーワードの集合から 2.4.2 項の方法で作られた NFA にそれを適用した場合，

得られるDFAの状態数は元のNFAの状態数より多くはならない．DFAの状態数は最悪の場合NFAの状態数の指数関数的になることを考えると，キーワード検索における上の状況は望ましいものであり，キーワードからまずNFAを設計してそれをDFAに変換する方法がよく使われる理由が頷ける．この場合，DFAの状態集合は以下のように構成すればよい．

a) NFAの開始状態をq_0とすると，$\{q_0\}$はDFAの一つの状態である．

b) pがNFAの状態で，開始状態q_0から$a_1a_2\cdots a_m$を読み込んだ結果pに到達するとする．このとき，以下に述べるNFAの状態の集合がDFAの一つの状態になる．

   1. q_0.
   2. p.
   3. NFAのその他の状態でq_0から$a_1a_2\cdots a_m$のある末尾（すなわち，$a_ja_{j+1}\cdots a_m$の形の列）を読み込んで到達できるすべての状態．

一般に，NFAの各状態pに対して上の(b)で述べたDFAの状態が一つ対応する．しかし，異なるpに対して構成した(b)の形の集合がたまたま一致することもあり得る．そういう場合が沢山あれば，それだけDFAの状態数は少なくなる．たとえば，もし二つのキーワードが同じ文字，たとえばa，で始まるとすると，NFAにおいてq_0からラベルaを持つ辺が二つあり，その行き先である二つのNFAの状態に対して上の(b)の方法で作ったDFAの状態は一致する．

例 2.15 図2.16のNFAから構成したDFAを図2.17に示す．この図では，NFAの状態pから上の規則(b)によって作られるDFAの状態は，図2.16でpのあったのと同じ位置に配置されている．たとえば，DFAの状態135を見て欲しい（ここで135は$\{1,3,5\}$の略記である）．この状態は，NFAの状態3を元に規則(b)で構成したものだが，このDFAの他の状態と同様にNFAの開始状態1を含み，さらに，状態5を含んでいる．その理由は，NFAの状態5は1から`e`で到達可能で，`e`は（1から3へ行くための入力列である）`we`の末尾の一つだからである．

この DFA の状態の遷移は，サブセット構成にしたがって求めることができる．すなわち，DFAのある状態SがNFAの開始状態q_0とそれ以外の状態$p_1,p_2,\ldots,p_n$から成るとき，Sから入力記号xに対するDFAの遷移の行き先

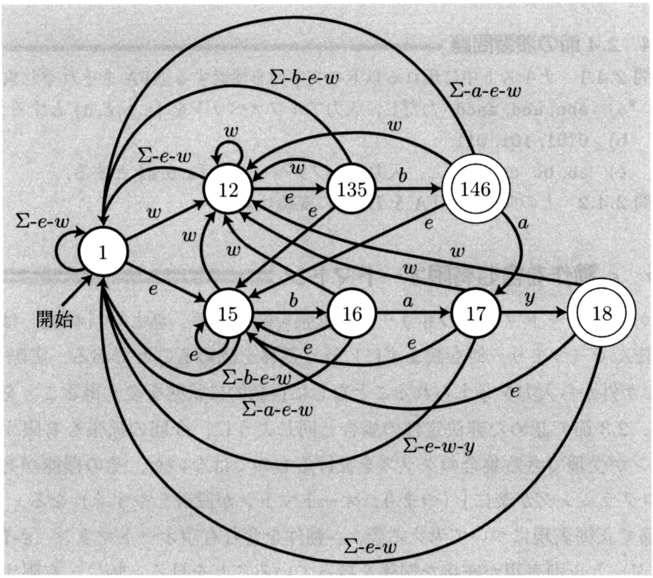


図 2.17　図 2.16 の NFA の DFA への変換

は，NFA において $q_0, p_1, p_2, \ldots, p_n$ から x で到達可能なすべての状態と q_0 から成る集合である．

たとえば，図 2.17 の状態 135 から入力記号 b による遷移を考えると，図 2.16 の NFA における状態 1, 3, 5 から記号 b による遷移として，1 から 1 へ，3 から 4 へ，5 から 6 へ行く三つの辺があるから，結局，DFA の状態 135 から記号 b による遷移の行き先は 146 となる．同じ状態から記号 e による遷移の行き先を考えると，NFA で状態 1, 3, 5 から記号 e による遷移は，1 から 1 自身と 5 に向かう辺があるだけだから，結局，DFA の状態 135 から記号 e による遷移の行き先は 15 となる．同様にして，135 から記号 w による遷移の行き先は 12 である．

その他の記号 x については，3 と 5 からの遷移はなく，1 から 1 へ向かう遷移があるだけだから，DFA の状態 135 から b, e, w 以外の Σ に属する記号による遷移の行き先は 1 である．図 2.17 でこれらの記号の集合を Σ − b − e − w で

表し，同様の記法を Σ から少数の記号を取り去った集合に対して用いている．

2.4.4 2.4 節の演習問題

♣ **問 2.4.1** テキスト中に現れる以下の文字列を検索する NFA をそれぞれ求めよ．

*a) abc, abd, aacd. ただし，入力アルファベットを $\{a, b, c, d\}$ とする．

b) 0101, 101, 011.

c) ab, bc, ca. ただし，入力アルファベットを $\{a, b, c\}$ とする．

♣ **問 2.4.2** 上の問題の NFA を DFA に変換せよ．

2.5 ε-動作を含む有限オートマトン

次に有限オートマトンのもう一つの拡張を紹介する．新しい「機能」は空列 ε を読んで（つまり，何も読まずに）行う遷移を認めることである．実際には，NFA が外から入力を与えられることなしに自発的に状態を変え得ることを意味する．2.3 節で認めた非決定性の場合と同じように，今回の拡張も有限オートマトンが受理できる集合のクラスを拡げるものではないが，その機能があると「プログラミングが楽に」（つまり，オートマトンが設計しやすく）なる．また，3.1 節で正則表現について考える際，ε-動作を含む有限オートマトン（**ε-NFA** と呼ぶ）と正則表現が密接な関係を持っていることを見る．特に，有限オートマトンで受理される言語のクラスと正則表現で表される言語のクラスが等しいことを示すのに ε-NFA は有用である．

2.5.1 ε-動作の使い方

まずはじめに，ε をラベルに持った遷移図を用いて ε-NFA を直観的に説明する．以下の例で，オートマトンは開始状態から受理状態に至る経路に沿ったラベルを繋いでできる列を受理するが，ただしラベルとして ε が書かれたとき，それは「見えない」列とする．すなわち，経路に沿ったラベルを繋いで列を作る際，ε があってもそれは無視する．

例 2.16 図 2.18 は，次のもので構成される 10 進数を受理する ε-NFA を表す．

(1) + または − の符号（これはなくてもよい）．

(2) 0 から 9 までの数字の列．

(3) 小数点．

(4) 数字のもう一つの列．この列か (2) の列のどちらかは空列でもよいが，少なくともどちらか一方は空でない列とする．

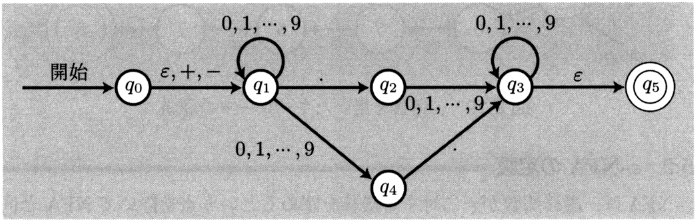


図 2.18 10 進数を受理する ε-NFA

ここで注意してほしいのは，q_0 から q_1 へは ε, $+$, $-$ のいずれかを読んだとき遷移することである．ここで状態 q_1 は，もしあれば符号と更にいくつか（0 個以上）の数字を処理して小数点は未処理という状況を表している．状態 q_2 は，小数点を見終わった状況を表すが，その前に数字が一つ以上あったかどうかは問わない．一方，q_4 は数字を一つ以上見てこれから小数点を読もうとしている状況を表す．状態 q_3 は小数点と（その前か後に）少なくとも一つ以上の数字を読んだことを表している．その後にまだ数字が続いていれば状態 q_3 でそれらを読み続けてもよいし，数字の列は終わったものと「見なし」て，何も読まずに受理状態 q_5 に進んでもよい． ▨

例 2.17 例 2.14 でキーワードの集合を認識する NFA の構成法は，もし ε-動作が許されれば簡潔になると前に述べた．たとえば，キーワード `web` と `ebay` を認識する NFA を図 2.16 で見たが，これは ε-動作を使うと，図 2.19 のようになる．一般に，各キーワードに対してあたかもそれがオートマトンで認識すべき唯一のキーワードであるかのように完全な状態の列を用意し，次いで，新しい状態（図 2.19 の状態 9）を加え，そこから各キーワードのために作ったオートマトンの開始状態に ε-遷移を加える． ▨

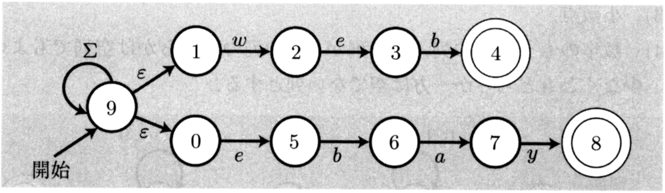


図 2.19　ε-遷移を用いたキーワードの認識

2.5.2　ε-NFA の定義

ε-NFA は，遷移関数が ε に対する遷移を認めるという点を除いて NFA と同様に定義することができる．正確にいうと，ε-NFA A の定義は $A = (Q, \Sigma, \delta, q_0, F)$ により与えられる．ここで，A の各要素は NFA の場合と同じで，ただし，δ の引数は次の二つである．

(1)　Q の要素

(2)　$\Sigma \cup \{\varepsilon\}$ の要素，すなわち入力記号かまたは空列 ε（ただし，ε は集合 Σ には属さないものとする）

例 2.18　図 2.20 の ε-NFA の場合，上の定義にしたがって書くと

$$E = (\{q_0, q_1, \ldots, q_5\},\ \{., +, -, 0, 1, \ldots, 9\},\ \delta,\ q_0,\ \{q_5\})$$

となる．ただし，δ は図 2.20 の遷移表の通りである．　▨

	ε	$+, -$	$.$	$0, 1, \cdots, 9$
$\to q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
$*q_5$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

図 2.20　図 2.18 に対する遷移表

2.5.3　ε-閉　包

以下で，ε-NFA に対する遷移関数の拡張を定義し，ε-NFA による文字列の受理と ε-NFA の言語を厳密に定義した上で，ε-NFA はなぜ DFA と同等の能

力を持つかを説明する．しかし，その前に準備として，ε-NFA の状態の**ε-閉包** (ε-closure) を定義する．直観的にいうと，状態 q の ε-閉包とは，q から ラベルが ε である遷移（以後，**ε-遷移**という）のみを伝って到達できる状態全体の集合を意味する．しかし，それを求めるには，q から ε-遷移で別の状態に到達したら，そこからまた ε-遷移で移れる状態へと次々に辿って行き，最終的に q からラベルが ε の辺だけから成る経路の行く先をすべて探しだす必要がある．正確には，各状態 q に対して q の ε-閉包 ECLOSE(q) を次で定義する．

基礎：状態 q は ECLOSE(q) に属する．

再帰：状態 p が ECLOSE(q) に属し，かつ状態 p から状態 r に ε-遷移があるとき r は ECLOSE(q) に属する．すなわち，δ が今考えている ε-NFA の遷移関数で p が ECLOSE(q) に属するとき，$\delta(p,\varepsilon)$ に属するすべての状態を ECLOSE(q) は含む．

例 2.19　図 2.18 のオートマトンで，各状態の ε-閉包は，次の二つの例外を除いてその状態自身のみから成る．例外の一つは，ECLOSE$(q_0) = \{q_0, q_1\}$ であり，もう一つは ECLOSE$(q_3) = \{q_3, q_5\}$ である．実際，この ε-NFA には ε-遷移が二つあり，その一方によって q_1 が ECLOSE(q_0) に加わり，他方により q_5 が ECLOSE(q_3) に加わる．

より複雑な場合を図 2.21 に示す．この図の場合，

$$\mathrm{ECLOSE}(1) = \{1,2,3,4,6\}$$

となる．つまり，ここに挙げた各状態に対して，状態 1 からその状態に至る経路で，ラベルが ε だけから成るものがある．たとえば，状態 6 へは $1 \to 2 \to 3 \to 6$ という経路がある．しかし，状態 7 は，1 から到達することはできるが，その際ラベルが ε でない辺 $4 \to 5$ を通らなければならないため，7 は ECLOSE(1) に

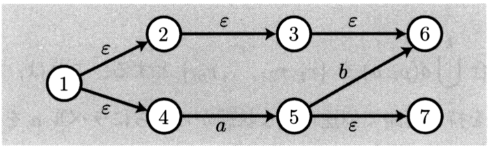


図 2.21　いくつかの状態とその遷移

属さない．なお，状態1から6へは $1 \to 4 \to 5 \to 6$ という ε-遷移でないものを含む経路も存在するが，そのことは今問題ではない．状態6がECLOSE(1)に入るか否かは，すべてのラベルが ε である経路が少なくとも一つ存在するか否かにかかっているのである． ▨

2.5.4 遷移関数の拡張と ε-NFA の言語

ε-NFA に（必ずしも空でない）入力列が与えられたとき，どんな遷移が行われるかを知る上で ε-閉包は役立つ．それを基に ε-NFA が入力を受理するとはどういうことかを定義する．

$E = (Q, \Sigma, \delta, q_0, F)$ を ε-NFA とする．はじめに，入力列が ε-NFA に与えられたときどういうことが起きるかを表現するため，遷移関数 δ の拡張 $\hat{\delta}$ を定義する．我々の意図は，状態 q から，w をラベルに持つ経路を通って到達できる状態全体の集合を $\hat{\delta}(q, w)$ が表すようにすることである．ただし，ε-NFA の経路のラベルが w であるとは，その経路を構成する遷移のラベル（入力記号または ε）を繋いだ結果が w であることを意味する．その場合，ε はもちろん空列として扱われ，w を構成する入力記号とは見なされない．$\hat{\delta}$ の再帰的な定義は次の通りである．

基礎：$\hat{\delta}(q, \varepsilon) = \mathrm{ECLOSE}(q)$．すなわち，もし経路のラベルが ε なら，ε をラベルとして持つ辺のみを辿って q から到達できる状態の全体を $\hat{\delta}(q, \varepsilon)$ は意味するが，それは $\mathrm{ECLOSE}(q)$ そのものである．

再帰：w が xa の形で，ただし a は w の末尾の記号とする（a は Σ に属し，ε ではない）．このとき，$\hat{\delta}(q, w)$ を次のように定義する．

(1) $\hat{\delta}(q, x)$ を $\{p_1, p_2, \ldots, p_k\}$ とする．すなわち，状態 q からラベルが x である経路を辿って到達可能な状態の全体が $p_1, p_2, \ldots, p_k$ だとする．これらの経路の最後のいくつかのステップが ε-遷移（ε をラベルとして持つ遷移）の場合もあり得るし，経路のそれ以外の部分にも ε-遷移はあるかも知れない．

(2) 和集合 $\displaystyle\bigcup_{i=1}^{k} \delta(p_i, a)$ を $\{r_1, r_2, \ldots, r_m\}$ とする．これは，状態 q からラベル x を持つ経路で到達可能な状態からさらにラベル a を持つ遷移で到達可能な状態の集合である．すると，これらの状態 r_j は，状態 q からラ

ベル w を持つ経路で到達可能な状態であるが，これらがそのすべてであるとは必ずしもいえない．というのは，これらの r_j からさらにいくつかの ε-遷移で辿れる状態がまだここには含まれていないからである．それを考慮したのが次の (3) である．

(3) $\hat{\delta}(q, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$ とおく．このように最後に閉包をとる操作を加えることにより，状態 q からラベル w を持つ経路で到達可能な状態のすべてが（最後に ε-遷移によって到達できる状態も含めて）集められたことになる．

例 2.20 図 2.18 の ε-NFA における $\hat{\delta}(q_0, \texttt{5.6})$ を計算してみよう．

- $\hat{\delta}(q_0, \varepsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$.
- $\hat{\delta}(q_0, \texttt{5})$ の計算は次の通りである．
  - (1) はじめに，上で求めた $\hat{\delta}(q_0, \varepsilon)$ の要素 q_0, q_1 から 5 を入力されて遷移する先を求めると，$\delta(q_0, \texttt{5}) \cup \delta(q_1, \texttt{5}) = \{q_1, q_4\}$ となる．
  - (2) 次に，上で求めた集合の各要素の ε-閉包を求めると，$\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$．これが求める $\hat{\delta}(q_0, \texttt{5})$ である．この (1) と (2) を次の二つの入力記号に対しても繰り返す．
- $\hat{\delta}(q_0, \texttt{5.})$ の計算は次の通りである．
  - (1) はじめに，$\delta(q_1, \texttt{.}) \cup \delta(q_4, \texttt{.}) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$ を求める．
  - (2) 次に，$\hat{\delta}(q_0, \texttt{5.}) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$ を求める．
- $\hat{\delta}(q_0, \texttt{5.6})$ の計算は次の通りである．
  - (1) はじめに，$\delta(q_2, \texttt{6}) \cup \delta(q_3, \texttt{6}) \cup \delta(q_5, \texttt{6}) = \{q_3\} \cup \{q_3\} \cup \varnothing = \{q_3\}$ を求める．
  - (2) 次に，$\hat{\delta}(q_0, \texttt{5.6}) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$ を求める．▨

以上の定義の下で，ε-NFA の言語を $L(E) = \{\ w \mid \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$ により定義する．すなわち，E の言語は，E の開始状態を少なくとも一つの受理状態に導くような入力列 w 全体から成る集合である．たとえば，上の例 2.20 で $\hat{\delta}(q_0, \texttt{5.6})$ は受理状態 q_5 を含むから 5.6 はこの ε-NFA E の言語 $L(E)$ に属する．

2.5.5 ε-遷移の除去

ε-NFA E が与えられたとき，E と同じ言語を受理する DFA D を構成することができる．その構成法は先に述べたサブセット構成とよく似ていて，E の状態集合の部分集合を D の状態とする．両者の違いは，E の場合，ε-遷移を考慮しなければならないことだが，そのために先に定義した ε-閉包が役立つ．

$E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ を ε-NFA とする．そのとき，E と等価な DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

は以下で定義される．

(1) Q_D は Q_E の部分集合全体から成る集合である．より詳しくいうと，その中で D において到達可能な状態は，ε-遷移に関して閉じている Q_E の部分集合（すなわち，Q_E の部分集合 S で，S に属する状態から ε-遷移だけを通って到達できる状態はみな S に属するもの，式で書くと $S = \bigcup_{q \in S} \mathrm{ECLOSE}(q)$ を満たす S）に限られる．

(2) $q_D = \mathrm{ECLOSE}(q_0)$．すなわち，$E$ の開始状態の ε-閉包を D の開始状態とする．この定義はサブセット構成とは異なることに注意せよ（NFA から DFA を構成するためのサブセット構成では，NFA の開始状態だけから成る単元集合を DFA の開始状態とした）．

(3) F_D は E の状態の集合で，受理状態を少なくとも一つ含むもの全体から成る集合である．すなわち，

$$F_D = \{\ S \mid S \text{ は } Q_D \text{ に属し，かつ } S \cap F_E \neq \varnothing \text{ を満たす }\}$$

(4) $\delta_D(S, a)$ は Q_D の要素 S と Σ の要素 a に対して次のように計算される．

(a) $S = \{p_1, p_2, \ldots, p_k\}$ とする．

(b) $\bigcup_{i=1}^{k} \delta_E(p_i, a)$ を求め，その結果を $\{r_1, r_2, \ldots, r_m\}$ とする．

(c) このとき，$\delta_D(S, a) = \bigcup_{j=1}^{m} \mathrm{ECLOSE}(r_j)$．

例 2.21 図 2.18 の ε-NFA を E として，この E から ε-遷移を除去して E と同等な DFA を構成すると図 2.22 のようになる．ただし，繁雑さを避けるため図 2.22 では，死状態 $\varnothing$ とそこへ向かう遷移を省略した．すなわち，図 2.22 の各状態 q について，もしそこから入力記号 a をラベルに持つ遷移がなければ，

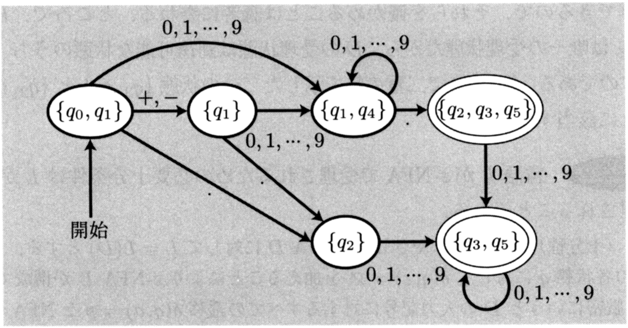


図 2.22　図 2.18 から ε-遷移の除去によって得られた DFA D

その状態 q から $\varnothing$ へ向けてラベル a を持つ遷移があるものとせよ．また，$\varnothing$ から $\varnothing$ へ向けてあらゆる入力記号をラベルに持つ遷移があるものとせよ．

E の開始状態は q_0 だから，D の開始状態は ECLOSE(q_0) すなわち $\{q_0, q_1\}$ である．最初にすべきことは，E で状態 q_0 と q_1 から Σ の入力記号（ここでは $+$ と $-$ の符号と，小数点 . と，0 から 9 までの数字）を読んでどこに行くかを調べることである．状態 q_1 から $+$ か $-$ を読んで進む先はないが，状態 q_0 からは q_1 に進む．したがって，$\delta_D(\{q_0, q_1\}, +)$ を求めるには，$\{q_1\}$ の ε-閉包を求めればよい．ところで，q_1 から始まる ε-遷移はないから，$\delta_D(\{q_0, q_1\}, +) = \{q_1\}$ となる．同様に，$\delta_D(\{q_0, q_1\}, -) = \{q_1\}$ となる．これらは図 2.22 で一本の辺としてまとめて示されている．

次に，$\delta_D(\{q_0, q_1\}, .)$ を求めるには，図 2.18 で小数点をラベルに持つ遷移が q_0 からはなく q_1 からは q_2 へ進むから，$\{q_2\}$ の ε-閉包を求めればよい．しかし，q_2 から始まる ε-遷移はないから，$\{q_2\}$ の ε-閉包は自分自身である．したがって，$\delta_D(\{q_0, q_1\}, .) = \{q_2\}$ である．

最後に，$\{q_0, q_1\}$ で数字を 1 文字読んだときの遷移を代表して，$\delta_D(\{q_0, q_1\}, 0)$ を考える．数字を読んだとき，q_0 から出発する遷移はないが，q_1 からは q_1 と q_4 へ進む．そしてこれらの二つの状態から始まる ε-遷移はないから，結局 $\delta_D(\{q_0, q_1\}, 0) = \{q_1, q_4\}$ となる．他の数字の場合も同様である．

以上で，図 2.22 の状態 $\{q_0, q_1\}$ から始まる遷移は説明した．他の遷移も同様

に計算できるので，それらを確かめることは読者に委ねる．ところで，E において q_5 は唯一の受理状態だから，D の受理状態は到達可能な状態のうち，q_5 を含むものである．図2.22で二重の輪で示した二つの状態 $\{q_3, q_5\}$ と $\{q_2, q_3, q_5\}$ がそれに該当する． ▨

定理 2.22 言語 L が ε-NFA で受理されるための必要十分条件は L が DFA で受理されることである．

証明 （十分性）この方向はやさしい．DFA D に対して $L = L(D)$ とする．このとき D の各状態 q に対して $\delta(q, \varepsilon) = \varnothing$ を加えることにより ε-NFA E が構成できる．なお，厳密にいうと D の入力記号に対するすべての遷移 $\delta(q, a) = p$ を NFA の遷移の形に $\delta(q, a) = \{p\}$ と書き換える必要がある．こうすると，E と D の遷移は全く同じで，E には ε-遷移がないことが明示的に述べられている．

（必要性）$E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ を ε-NFA とする．これに，2.5.5 項の初めに述べたサブセット構成の修正版を適用して DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ を求める．このとき $L(E) = L(D)$ であることを示せばよいが，そのために E と D の遷移関数の拡張が等しいことを示せばよい．正確には，各入力列 w に対して $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ であることを w の長さに関する帰納法で示す．

基礎：$|w| = 0$ のとき，$w = \varepsilon$ である．このとき，定義から $\hat{\delta}_E(q_0, \varepsilon) = \mathrm{ECLOSE}(q_0)$ かつ $q_D = \mathrm{ECLOSE}(q_0)$ である．また，DFA ではすべての状態 p について $\hat{\delta}(p, \varepsilon) = p$ であるから，特に $\hat{\delta}_D(q_D, \varepsilon) = q_D = \mathrm{ECLOSE}(q_0)$．よって，$\hat{\delta}_E(q_0, \varepsilon) = \hat{\delta}_D(q_D, \varepsilon)$ であることが示された．

帰納：w が xa の形で，a は w の末尾の記号であり，x については先の等式が成り立つとする．すなわち $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x)$ が成り立つとする．この両辺で表される集合を $\{p_1, p_2, \ldots, p_k\}$ とする．そのとき，ε-NFA に対する $\hat{\delta}$ の定義から，$\hat{\delta}_E(q_0, w)$ は以下のように計算される．

(1) $\displaystyle\bigcup_{i=1}^{k} \delta_E(p_i, a)$ を $\{r_1, r_2, \ldots, r_m\}$ とする．

(2) このとき，$\displaystyle\hat{\delta}_E(q_0, w) = \bigcup_{j=1}^{m} \mathrm{ECLOSE}(r_j)$．

一方，DFA D の作り方（サブセット構成の修正版）から，$\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ は上の (1),(2) と同じステップで構成される．したがって，$\hat{\delta}_D(q_D, w) = \delta_D(\hat{\delta}_D(q_D, x), a) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ は $\hat{\delta}_E(q_0, w)$ と等しい．以上で，$\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ であることが示された． ▨

2.5.6 2.5 節の演習問題

♣ !問 2.5.1 次の ε-NFA を考える.

a) 各状態の ε-閉包を計算せよ.

b) このオートマトンで受理される長さ 3 以下のすべての入力列を求めよ.

c) このオートマトンを DFA に変換せよ.

	ε	a	b	c
$\rightarrow p$	$\varnothing$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\varnothing$
$*r$	$\{q\}$	$\{r\}$	$\varnothing$	$\{p\}$

♣ 問 2.5.2 前問を次の ε-NFA に対して行え.

	ε	a	b	c
$\rightarrow p$	$\{q,r\}$	$\varnothing$	$\{q\}$	$\{r\}$
q	$\varnothing$	$\{p\}$	$\{r\}$	$\{p,q\}$
$*r$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

♣ 問 2.5.3 次の言語に対する ε-NFA を設計せよ. 答がなるべく簡潔になるように ε-遷移を上手く利用せよ.

a) 0 個かそれ以上の a が並び, その後に 0 個かそれ以上の b が並び, さらにその後に 0 個かそれ以上の c が並んだ列の全体.

!b) 01 が 1 回以上繰り返された列と 010 が 1 回以上繰り返された列の全体.

!c) 0 と 1 の列で, 少なくとも末尾の 10 文字のうちの一つは 1 であるものの全体.

2.6 第 2 章 の 要 約

◆**決定性有限オートマトン**: DFA は状態の有限集合と入力記号の有限集合を持つ. 状態の一つが開始状態として指定され, いくつか (0 個以上) の状態が受理状態として指定される. 入力記号が読み込まれるたびに状態がどのように変化するかを表すのが遷移関数である.

◆**遷移図**: オートマトンをグラフの形で表すと都合がよい. ただし, グラフの頂点でオートマトンの状態を表し, 頂点間を結ぶ辺に入力記号をラベルとして書き添えたものでオートマトンの状態遷移を表す. 開始状態は矢印で示し, 受理状態は二重の輪で表す.

◆**オートマトンの言語**: オートマトンは記号列を受理する. オートマトンが開始状態から出発して入力記号列を先頭から順に 1 文字ずつ読みながら指定された状態

遷移を続けて行くとやがて受理状態に到達するとき，その記号列は受理される．遷移図の言葉でいうと，記号列がオートマトンで受理されるのは，それが開始状態からある受理状態に至る経路のラベルと一致するときである．

◆**非決定性有限オートマトン**：NFA が DFA と異なるのは，NFA の場合，現在の状態と入力記号に対して次の状態の候補がいくつあってもよい（0 個でもよい）点である．

◆**サブセット構成**：NFA の状態の集合を DFA の 1 個の状態と見なすことによって，どんな NFA もそれと同じ言語を受理する DFA に変換することができる．

◆**ϵ-遷移**：空列に対する状態遷移（すなわち，何も入力記号を読まずに状態を変えること）を許すことによって，NFA の概念を拡張することができる．このように拡張された NFA についても，それと同じ言語を受理する DFA がある．

◆**テキスト検索への応用**：非決定性有限オートマトンは，テキストファイル中にあるいくつかの指定したパターンがあるか否かを検索する際の考え方を表現するのに適している．これらのオートマトンはソフトウェアで直接シミュレートするか，あるいはまず DFA に変換してそれからシミュレートしてもよい．

2.7 第2章の参考文献

有限状態を持つシステムの研究は文献 [2] によって始められたとされている．しかし，この文献は計算を神経網モデルに基づいて研究したもので，現在よく知られている有限オートマトンに基づくものではない．伝統的な DFA の概念は，互いに若干の違いのあるいくつかの形で，ただし互いに独立に [1], [3], [4] によって提案された．非決定性有限オートマトンとサブセット構成は [5] による．

[1] D. A. Huffman, “The synthesis of sequential switching circuits,” *J. Franklin Inst.* **257**:3-4 (1954), pp.161–190 and 275–303.

[2] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervious activity,” *Bull. Math. Biophysics* **5** (1943), pp.115–133.

[3] G. H. Mealy, “A method for synthesizing sequential circuits,” *Bell System Technical Journal* **34**:5 (1955), pp.1045–1079.

[4] E. F. Moore, “Gedanken experiments on sequential machines,” in [6], pp.129–153.

[5] M. O. Rabin and D. Scott, “Finite automata and their decision problems,” *IBM J. Research and Development* **3**:2 (1959), pp.115–125.

[6] C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956.

3 正則表現と正則言語

この章では最初に「正則表現」と呼ばれる概念を紹介する．すでに1.1.2項でその簡単な例を見たが，正則表現はオートマトンとは異なる考え方で言語を正確に表現するための記法である．またそれと同時に，正則表現を一種のプログラミング言語と見ることもできる．この後者の観点における重要な応用例として，テキスト検索とコンパイラの中で正則表現が果たす役割について述べる．正則表現は非決定性有限オートマトンと密接な関係にあり，ソフトウェアの一部を記述する上で「ユーザに優しい」NFAと見ることができる．

以下では，正則表現を定義した後，それがすべての正則言語（すなわち，オートマトンで認識できる言語）を，そしてそれのみを表現できることを示す．次いで，いくつかのソフトウェアで正則表現が実際にどのように使われているかを述べた後，正則表現について成り立つ代数的な性質を調べる．それらは，算術における代数的性質と多くの類似点をもつと同時に，いくつかの重要な相異点をもつ．

3.1 正 則 表 現

前章では，言語に対する機械風の記述である決定性および非決定性有限オートマトンについて述べたが，この章では言語に対する代数的な記述である「正則表現」を取り上げる．そして，種々のオートマトンで定義される言語と同じものが正則表現で表現できることを見る．ただし，オートマトンはある種の記号列を認識する仕組みを提供するのに対して，正則表現はそれらの記号列の形そのものを記述する点が異なる．このため，正則表現は記号列を処理する以下のようなシステムで，入力形式を指定するために利用されている．

(1) Unixの grep のような検索コマンド，あるいはウェブのブラウザや文書

整形システムにおける同様の検索コマンド．これらのシステムでは，ユーザがファイル中の検索したい記号列のパターンを記述する際，正則表現と似た記法を取り入れている．そして検索システムでは，そのような表現を DFA か NFA に変換し，検索すべきファイルをそのオートマトンに与えて処理を行う．

(2) コンパイラにおける字句解析（lexical analysis，語彙解析ともいう）のためのプログラムを自動生成する `Lex` や `Flex` などのコマンド．字句解析とは，ソースプログラムを**トークン**（token）と呼ばれる論理的な最小単位に分解する仕事を行うコンパイラの部分を指す．トークンには，予約語（たとえば，`while`）や，名前（たとえば，英字の後に英字か数字がいくつか続いた列）や，`+` や `<=` のような記号などがある．字句解析プログラムの自動生成システムでは，そのコンパイラで扱おうとするすべてのトークンの形式を基本的に正則表現の形で書いたものを受け取り，それらのトークンを識別するための DFA を作る．

3.1.1 正則表現の演算

一つの正則表現は一つの言語を表す．簡単な例をあげると，**01* + 10***は，0 の後に任意個の 1 が続いた列と，1 の後に任意個の 0 が続いた列から成る言語を表す正則表現である．このような正則表現の解釈の仕方についてはすぐ後で説明するが，それに先立ち，正則表現で用いる三つの演算を紹介する．

(1) 二つの言語 L と M の**集合和**（union）$L \cup M$ は，L か M（またはその両方）に含まれる列全体から成る集合である．たとえば，$L = \{001, 10, 111\}$ および $M = \{\varepsilon, 001\}$ のとき，$L \cup M = \{\varepsilon, 10, 001, 111\}$．

(2) 言語 L と M の**連接**（concatenation）は，L に属する列を一つとり，その後に M に属する列を連接することによってできる列全体から成る集合である．ただし，二つの列の連接は 1.5.2 項で定義した通りである．すなわち，一つの列の直後にもう一つの列を繋げてできる列を二つの列の連接という．言語の連接を表すのに · を用いるか，または（連接の演算はしばしば「ドット」演算といわれるにも拘らず）演算記号を省略することもある．たとえば，$L = \{001, 10, 111\}$ かつ $M = \{\varepsilon, 001\}$ のとき，$L \cdot M$ または LM は $\{001, 10, 111, 001001, 10001, 111001\}$ である．LM の中の

最初の三つの列は L の要素と ε を連接した結果である．ε は連接の単位元であるから，今述べた三つの列は L の要素そのものである．一方，LM の残りの三つの列は L の要素と M のもう一つの要素 001 を連接した結果である．たとえば，L の要素 10 と M の要素である 001 を連接した結果として LM の要素 10001 が得られる．

(3) 言語 L の**閉包**（または**スター**または**クリーネ**（**Kleene**）**閉包**[1]ともいう）は，L の中から有限個の列を重複を許して取り出し（すなわち，同じものを何度も繰り返しもってきてもよい），それらを連接してできる列全体の集合であり，L^* と書く．たとえば，$L = \{0,1\}$ のとき，L^* は 0 と 1 から成る列の全体である．$L = \{0,11\}$ のとき L^* は，0 と 1 から成る列で 1 が常に二つずつ組になって並んでいるものの全体であり，たとえば $011, 11110, \varepsilon$ などが L^* の中に含まれ，$01011, 101$ は含まれない．正確にいうと，L^* は無限個の集合の和 $\cup_{i\geq 0} L^i$ を表す．ただし，$L^0 = \{\varepsilon\}$，$L^1 = L$，かつ $i > 1$ のとき L^i は L を i 個連接した結果 $LL \cdots L$ を表す．

例 3.1 言語の閉包の考え方は少しわかりにくいかも知れないので，いくつか例を示す．まず，$L = \{0,11\}$ とする．L^0 は L の如何に関わらず常に $\{\varepsilon\}$ である．というのは，L の 0 乗は L の中から 0 個の列をとって連接した結果から成る集合だが，0 個の列を並べた結果は空列 ε だからである．$L^1 = L$ は，L の中から一つの列をとって連接した結果（つまり，今取り出した列そのもの）の全体である．このように，L^* を表す無限和 $\cup_{i\geq 0} L^i$ のうちの最初の 2 項の部分和 $L^0 \cup L^1$ は $L = \{0,11\}$ のとき $\{\varepsilon, 0, 11\}$ である．

次に L^2 であるが，そのためにまず $L = \{0,11\}$ から重複を許して二つの列を取り出す方法が全部で（最初の列が 0 か 11 か，次の列が 0 か 11 かの）4 種類あることに注意する．そのようにして取り出した二つの列を連接した結果の全体が L^2 であるから，$L^2 = \{00, 011, 110, 1111\}$ である．同様に L^3 は L の中から三つの列を取り出して並べた結果の全体であるから

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

---

[1] 「クリーネ閉包」という呼び名は正則表現という記法とこの演算とを最初に考えた S.C.Kleene に因む名前である．

となる．L^* を求めるには各 i に対して L^i を計算してそれらの全体の集合和をとる必要がある．上の例の場合，各 i について L^i は 2^i 個の要素を持つ．一般に L が有限集合のとき各 L^i は有限集合だが，それらの無限個の和は（例外的な場合を除いて）無限集合となる．

次に，0 を任意個並べてできる列の全体を L とする．このとき，L はそれ自身無限集合であるが，L^* はどんな集合になるだろうか．まず，L の如何によらず $L^0 = \{\varepsilon\}$ かつ $L^1 = L$ である．L^2 は 0 だけから成る任意の列を一つとり，同じく 0 だけから成る任意の列をもう一つ取って，それらを連接した結果の全体であるが，そのようにして得られた列はやはり 0 だけから成る．そして，0 だけから成るどんな列も，0 だけから成る二つの列を連接した結果として表される（ε は 0 を 0 個並べてできる列であり，これを上の二つの列のどちらかとしてとり得ることに注意せよ）．こうして $L^2 = L$ であることがわかる．同様にして $L^3 = L$，以下同様である．このようにして無限和 $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$ は，L が 0 だけから成る列全体の場合，L 自身に等しい．

最後の例は $\varnothing^* = \{\varepsilon\}$ である．この場合，$\varnothing^0 = \{\varepsilon\}$ だが，$i \geq 1$ のとき $\varnothing^i$ は（空集合から要素を取りだすことが不可能なため）空集合である．実際，$\varnothing$ は，閉包が無限集合でない二つの例外のうちの一つである．▨

3.1.2　正則表現の構成

様々な代数の定義は，ふつう，定数や変数などの式の定義から始まる．そのような単純な式から出発して，ある種の演算を繰り返し適用することにより，次第に複雑な式が代数の対象として構成される．たとえばよく知られる算に対す

スター演算の使用

スター演算が本書で最初に登場したのは 1.5.2 項で，アルファベット Σ に対する Σ^* という形で登場した．そこではアルファベット Σ から任意に記号を選んで並べてできる列の全体を Σ^* と書いた．閉包演算は本質的にこれと同じだが，型に関する若干の相異がある．

今 L を長さが 1 の列全体から成る言語（すなわち，各記号 a を長さ 1 の列と見たものの全体が L である）とする．このとき，L と Σ とは「同じ」ように見えるが，厳密にいうと，L は列の集合であるのに対して Σ は記号の集合であるという点で，両者は「型」が異なる．しかし，L^* は Σ^* と同じ言語を表す．

る代数の場合，整数や実数などの定数と変数から出発して，それらに $+$ や $\times$ などの演算記号を使ってより複雑な式が構成される．なお，その際いくつかの演算対象をまとめるために括弧が使われることもある．

正則表現も一種の代数として上の方法にしたがって構成される．すなわち，ある特定の簡単な言語を表す「定数」と，(いろいろな言語を表す)「変数」から始めて，3.1.1 項で述べた三つの（言語に対する）演算——集合和とドットとスター——を繰り返し適用してより複雑な表現を構成する．このことは，以下のように再帰的定義の形で正確に述べることができる．我々はこの定義で，正則表現とは何かだけでなく，それぞれの正則表現 E が表す言語 $L(E)$ も同時に定義する．

基礎：次の三つの場合から成る．

(1) 定数 ε と $\varnothing$ は正則表現であり，それぞれ言語 $\{\varepsilon\}$ と空集合 $\varnothing$ を表す．すなわち，$L(\varepsilon)=\{\varepsilon\}$ および $L(\varnothing)=\varnothing$ である．

(2) a が記号のとき，$\mathbf{a}$ は正則表現であり，言語 $\{a\}$ を表す．すなわち，$L(\mathbf{a})=\{a\}$ である．このように我々は，記号 a とそれに対する正則表現 $\mathbf{a}$ を太字を使って区別する．

(3) 任意の言語を表す変数（通常，L のように大文字の斜体を使う）を正則表現の中で使うことがある[2)]．

再帰：この部分は，先に紹介した三つの演算に関するものと，括弧に関するものと全部で四つの場合から成る．

(1) E と F が正則表現のとき $E+F$ も正則表現であり，それが表す言語は $L(E)$ と $L(F)$ の集合和である．すなわち，$L(E+F)=L(E)\cup L(F)$.

(2) E と F が正則表現のとき EF も正則表現であり，それが表す言語は $L(E)$ と $L(F)$ の連接である．すなわち，$L(EF)=L(E)L(F)$. なお，ここでは連接を表す演算記号を省略したが，たとえば，**01** と書く代わりに

---

2) ＜**訳注**：本書で「正則表現」というとき，普通はその中に変数を含まないものを指す．したがって，この基礎のステップの (3) を普通は無視してよい．ただし，例外として 3.4 節では，上の (2) の “記号” の代わりに (3) の “変数” を含む正則表現が，普通の（変数を含まない）正則表現と共に登場する．そのため 3.4 節ではこの両者を区別する意味で，前者を「変数を含む正則表現」と呼び，後者を単に「正則表現」または「具体的な正則表現」と呼ぶ．＞

表現とその言語

厳密にいうと，正則表現 E は言語に対する表記法であって言語そのものではない．E が表す言語はしたがって E ではなく $L(E)$ と書くべきである．しかし現実には，$L(E)$ の代わりに E と書くことが多い．我々も，誤解を生じない場合はこの慣習にしたがう．

$\mathbf{0{\cdot}1}$ のようにドットを書いてもよい．しかし，我々は正則表現における連接演算ではドットを省略することにする[3]．

(3)　E が正則表現のとき E^* も正則表現であり，それが表す言語は $L(E)$ の閉包である．すなわち，$L(E^*) = (L(E))^*$.

(4)　E が正則表現のとき (E) も正則表現であり，それが表す言語は E が表す言語と同じである．すなわち，$L((E)) = L(E)$.

例 3.2　0 と 1 が交互に現れる列全体を表す正則表現を考えよう．そのためにまず，列 01 だけから成る言語の正則表現を考え，その後でスター演算を適用すると $0101\cdots01$ の形の列全体を表す正則表現が得られる．

正則表現の構成法の基礎より，$\mathbf{0}$ と $\mathbf{1}$ はそれぞれ言語 $\{0\}$ と $\{1\}$ を表す正則表現である．その二つの表現の連接 $\mathbf{01}$ は，言語 $\{0\}\{1\}$，つまり $\{01\}$ を表す正則表現である．一般に，ただ一つの列 w だけから成る集合 $\{w\}$ を表す正則表現として w を用いることがあるが，厳密にいうと w を構成する各記号を太字にすべきである．そのようなフォントの違いを無視すれば w は $\{w\}$ を表す正則表現といえる．

列 01 を何回でも（0 回以上）繰り返してできる列の全体は正則表現 $(\mathbf{01})^*$ で表される．ここで，$\mathbf{01}$ を括弧で括ったのは，そうしないと $\mathbf{0}(\mathbf{1}^*)$（0 の後に 1 がいくつか並んだ列の全体を表す正則表現）と解釈されるためである．このような正則表現の解釈は，演算の優先順位の決め方による．それについては後の 3.1.3 項で詳しく述べるが，簡単にいうと，スター演算はドット演算より結合の優先度が高いため，スター演算が連接より先に実行される．

ところで，$L((\mathbf{01})^*)$ は我々が表現したいと思っている言語そのものではない．

---

[3] UNIX の正則表現ではドットを（任意の ASCII 文字を表すという）別の目的のために使用している．

ここに含まれるのは，0 と 1 が交互に現れる列の中で 0 で始まり 1 で終わるものだけであり，その他に 1 で始まる列や 0 で終わるものがある．それらをすべて網羅するには，残る三つの場合を表す三つの正則表現を構成してその和をとればよい．実際，0 と 1 が交互に現れる列のうち，1 で始まり 0 で終わる列の全体は $(\mathbf{10})^*$ で表され，0 で始まり 0 で終わる列全体は $\mathbf{0}(\mathbf{10})^*$ で，1 で始まり 1 で終わる列全体は $\mathbf{1}(\mathbf{01})^*$ でそれぞれ表される．したがって求める 0 と 1 が交互に現れる列全体は，これら四つの言語の集合和として，次の正則表現で表される．

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

ここで，演算記号 + は集合和を表すことに注意せよ．

上と同じ集合を表す，もっと簡潔な正則表現もある．それを求めるには，最初は前と同じように $(\mathbf{01})^*$ から始める．この表現の前に $\varepsilon + \mathbf{1}$ を連接することにより，0 と 1 が交互に現れる列の中で 0 で始まり 1 で終わるものだけでなく，その前に 1 がついた列も含まれることになる．同様にして，さらに右側から $\varepsilon + \mathbf{0}$ を連接すると，末尾が 0 の列も加わる．実際，

$$L(\varepsilon + \mathbf{1}) = L(\varepsilon) \cup L(\mathbf{1}) = \{\varepsilon\} \cup \{1\} = \{\varepsilon, 1\}$$

であるから，この言語と任意の言語 L を連接すると，ε と L の要素との連接として L に属する列全体が得られ，1 と L の要素との連接として $1w$ の形の列（ただし w は L に属する列）が得られる．よって，0 と 1 が交互に現れる列全体を表すもう一つの正則表現として次が得られる．

$$(\varepsilon + \mathbf{1})(\mathbf{01})^*(\varepsilon + \mathbf{0})$$

なお，この表現における括弧は，演算の順序を示すために必要である． ▨

3.1.3 正則表現中の演算の順序

他の代数の場合と同様に，正則表現の場合もその中に現れる演算の結合に関する優先順位がきめられている．通常の算術演算における優先順位は，たとえば $xy + z$ の場合，積 xy の演算が先に行われその後で和がとられる（したがって，この表現は $(xy) + z$ を表し，$x(y + z)$ ではない）．また，同じ演算が続いて現れたときは左から先に実行する．（したがって，たとえば $x - y - z$ は $(x - y) - z$ の意味であって $x - (y - z)$ の意味ではない）．正則表現の場合，演算の順序は次のように決められている．

(1)　スター演算は優先順位が最も高い．すなわち，スター演算はその左にある最短の正則表現に対して適用される．

(2)　優先順位が次に高いのは連接（ドット演算）である．まずはじめにスター演算をすべて適用し，次いで連接演算を互いに隣り合っている正則表現同士の間で実行する．連接演算は結合法則を満たすから，隣り合った正則表現をどんな順序で連接してもよい（たとえば，$\mathbf{012}$ を $(\mathbf{01})\mathbf{2}$ のように左から順に連接してもよい）．

(3)　最後に，集合和（演算記号 +）を適用する．集合和の演算も結合法則を満たすから，連続して集合和をとる場合その順序を気にする必要はない（たとえば，左から順に適用すればよい）．

正則表現の中で，上に述べた優先順位にしたがわずに演算を実行したい場合も当然あり，その場合は，括弧を用いて自分で希望する順序を自由に指定することができる．決められた優先順位にしたがう場合でも，もしそうしたければ括弧を使って差し支えない．

例 3.3　正則表現 $\mathbf{01}^*+\mathbf{1}$ の場合，次の順序で演算が実行される：$(\mathbf{0}(\mathbf{1}^*))+\mathbf{1}$．まず，スター演算がそのすぐ左の $\mathbf{1}$ に対して適用される．次に $\mathbf{0}$ と $(\mathbf{1}^*)$ が連接されて $(\mathbf{0}(\mathbf{1}^*))$ となる．そして最後に今得られた結果とその右にある $\mathbf{1}$ に対して集合和の演算が実行される．

したがって，この表現の表す言語は，0 の後に 1 がいくつでも（0 個以上）続いた列と，列 1 から成る．ここで，もしスター演算より連接の方を先に実行したければ，括弧を $(\mathbf{01})^*+\mathbf{1}$ のようにつければよい．その場合表される言語は，01 を何回でも（0 回以上）繰り返した結果と列 1 から成る．また，もし連接より集合和を先に行いたければ，集合和の部分に括弧をつけて $\mathbf{0}(\mathbf{1}^*+\mathbf{1})$ とする．この表現の表す言語は，0 の後に 1 がいくつでも（0 個以上）続いた列の全体である．

3.1.4　3.1 節の演習問題

♣　**問 3.1.1**　次の言語に対する正則表現を求めよ．

*a)　アルファベット $\{a,b,c\}$ 上の文字列で，1 個以上の a と，1 個以上の b を含む列の全体．

b)　0 と 1 の列で，右端から 10 文字目が 1 であるものの全体．

c)　0 と 1 の列で，部分列として 11 を 2 回以上含まない列の全体．

♣ **!問 3.1.2**　次の言語に対する正則表現を求めよ．

*a)　0 と 1 の列で，その中に現れる 00 はどの 11 よりも左にある列の全体．

b)　0 と 1 の列で，その中の 0 の個数は 5 の倍数である列の全体．

♣ **!!問 3.1.3**　次の言語に対する正則表現を求めよ．

a)　0 と 1 の列で，部分列として 101 を含まないものの全体．

b)　同数の 0 と 1 から成る列で，その列のどこで切ってもその左側の部分について次の 2 条件が成り立つものの全体：0 の個数が 1 の個数より二つ多いことはなく，1 の個数が 0 の個数より二つ多いこともない．

c)　0 と 1 の列で，その中に含まれる 0 の個数は 5 の倍数で 1 の個数は偶数である列の全体．

♣ **!問 3.1.4**　次の正則表現の表す言語を英語（または日本語）で説明せよ．

*a)　$(\mathbf{1}+\varepsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$.

b)　$(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0}+\mathbf{1})^*$.

c)　$(\mathbf{1}+\mathbf{10})^*\mathbf{1}^*$.

♣***!問 3.1.5**　例 3.1 で，閉包が有限集合である二つの言語のうちの一つは空集合であると述べたが，残りの一つは何か．

3.2　有限オートマトンと正則表現

正則表現による言語の記述は，有限オートマトンによる記述と基本的に異なる考え方に基づいているが，この両者は同じ表現力を持つ（つまり，どちらか一方の方法で表せるものは他方でも表せる）ことが示される．我々は既に，決定性有限オートマトンと（ε-動作を許すものとそうでないものと）2 種類の非決定性有限オートマトンが，同じ言語のクラスを受理することを示した．正則表現もこれと同じクラスを定義することを示すために，以下で次のことを示す．

(1)　上に述べたどのオートマトンで定義される言語も正則表現で表現できること．それを示す際，DFA の受理言語について考えれば十分である．

(2)　正則表現で表される言語は上記のどれかのオートマトンで受理できること．それを示す際，その言語を受理する ε-動作つきの NFA があることを示すのが最も容易である．

図 3.1 に，我々がすでに証明したかまたはすぐ下で証明しようとしている言語のクラス間の関係をまとめて示す．この図で，クラス X から Y へ向かう辺

は，クラス X で定義される言語はすべてクラス Y でも定義できることを意味する．このグラフはこれらの辺で強く連結されている（すなわち，四つの頂点のどれからも他の頂点に移動できる）から，これらの辺がすべて示されれば四つのクラスはみな等しいことがわかる．

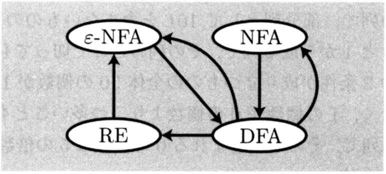


図 3.1　正則言語に対する四つの表記法が等価であることの証明手順

3.2.1 DFA から正則表現へ

与えられた DFA に対して，その言語を表す正則表現を作る方法は極めて技巧的である．その大まかな考え方は，DFA の遷移図上の経路のラベルの集合を表す正則表現を，簡単な経路から始めて次第に複雑なものへと段階的に（あるいは再帰的に）構成していく，というものである．ただし，ここで最も簡単な経路とは，出発点と終着点だけがあり，途中で経由する点がない経路をいい，最も複雑な経路とは，途中でどんな点を経由してもよい経路を意味する．つまり，ここで経路が簡単とか複雑とかいうのは，途中でどんな点を経由することができるかに関する制限の強弱を意味する．

定理 3.4　DFA の受理集合は正則表現で記述できる．すなわち，DFA A に対して $L = L(A)$ のとき，$L = L(R)$ を満たす正則表現 R がある．

証明　A の状態を $\{1, 2, ..., n\}$ とする．これは，A の状態の名前が元々どんなものであっても，それらは有限個なのでその個数を n とすると，状態の名前を付け変えて $1, 2, ..., n$ と呼ぶことができるという意味である．我々の最初の，そして最も難しい仕事は，簡単なものから複雑なものへと段階的に，正則表現の集まりを構成していくことである．

A の状態 i から j に至る経路で，k より大きな名前の状態を途中で経由しないもののラベル（つまり，その経路に含まれる辺のラベルを順に連接してできる列）の全体を表す正則表現 $R_{ij}^{(k)}$ を以下で構成する．ただし,「途中で経由する頂点」として経路の始点と終点は含まないものとする．

図 3.2 は，$R_{ij}^{(k)}$ で示される経路（つまり，$L(R_{ij}^{(k)})$ の要素）に対する条件を図解したものである．この図で垂直方向に A の状態の番号が（下から上に向かって状態 1 から n まで）示され，横方向は始点 i から経路に沿って終点 j まで進んでいく様子を示している．この図では，i と j が共に k より大きい場合を描いているが，その片方または両方が k 以下でもよい．また，この図では経路が状態 k を途中で 2 回実際に通っているが，k より大きい番号の状態を途中で（両端の i と j 以外の部分で）通らないことが重要である．

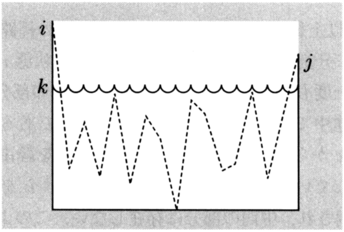


図 3.2　ラベルが正則表現 $R_{ij}^{(k)}$ の言語に属する経路

表現 $R_{ij}^{(k)}$ を構成するに当たって，我々は $k = 0$ から始めて最終的に $k = n$ まで，再帰的定義の考え方にしたがって構成していく．そのとき，特に $k = n$ の場合 $R_{ij}^{(n)}$ の言語は，状態 i から j に至る経路で n より大きい番号の状態を経由しない経路のラベルの集合であるが，n より大きい番号の状態はそもそも存在しないため，これは単に，状態 i から j に至る経路のラベルの全体に他ならないことに注意せよ．

基礎：基礎は $k = 0$ の場合である．A の状態の名前はすべて 1 以上であるから，経路が $R_{ij}^{(0)}$ の条件を満たすのは，その経路が途中で経由する点がない場合である．そのような条件を満たす経路は次の二つのいずれかである．

(1)　頂点 i から j に向かう辺（つまり，長さ 1 の経路）．

(2)　ただ一つの頂点 i のみから成る長さ 0 の経路．

もし，$i \neq j$ なら，上の (1) の場合しかあり得ない．このとき実際に何が (1) に該当するかを知るには，DFA A の遷移図で i から j へどんな入力 a を読んで遷移することができるかを見ればよい．

a)　そのような入力記号 a がなければ，$R_{ij}^{(0)} = \varnothing$.

b)　そのような入力記号 a が一つのときは，$R_{ij}^{(0)} = \mathbf{a}$.

c)　そのような入力記号 $a_1, a_2, ..., a_k$ が二つ以上あるときは，$R_{ij}^{(0)} = \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

一方 $i = j$ のときは，$R_{ij}^{(0)}$ の条件を満たす経路として上のものの他に長さ 0 の経路

がある．ところで，長さ 0 の経路のラベルから成る言語は（その経路に沿ったラベルの列も長さ 0 であるから）正則表現 ε で表される．したがって，$i=j$ のとき，$R_{ij}^{(0)}$ は上に述べた (a) から (c) のそれぞれに ε を加えればよい．すなわち，求める正則表現 $R_{ij}^{(0)}$ は，(a) の場合 ε，(b) の場合 $\varepsilon+\mathbf{a}$，(c) の場合 $\varepsilon+\mathbf{a}_1+\mathbf{a}_2+\cdots+\mathbf{a}_k$ である．

再帰：状態 i から j に至る経路で，k より大きい番号の状態は途中で通らないものを次の二つに分類して考える．

(1) 状態 k を 1 回も経由しない経路．この場合，その経路のラベルは $R_{ij}^{(k-1)}$ の言語に属する．

(2) 状態 k を 1 回以上経由する経路．この場合，その経路を図 3.3 に示すようにいくつかの部分に分割できる．その最初の部分は，状態 i から状態 k に至る経路で，途中で k を一度も経由しないものであり，最後の部分は，状態 k から状態 j に至る経路で，途中で k を一度も経由しないものである．そして中間の部分はいずれも，状態 k から k に至る経路で，途中で k を経由しないものである．もし，いま問題にしている元の経路が途中で一度だけ k を経由するなら，それは二つの部分に分割され，中間の部分は存在しない．このようにして分割された経路のラベルの全体は正則表現 $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^*R_{kj}^{(k-1)}$ によって表される．すなわち，最初の正則表現 $R_{ik}^{(k-1)}$ は上で述べた最初の部分（つまり，状態 i から初めて状態 k に到達するまでの部分）に対応し，2 番目の正則表現 $(R_{kk}^{(k-1)})^*$ は元の経路が最初に状態 k に到達してから最後に k に到達するまでの部分に対応し，最後の正則表現 $R_{kj}^{(k-1)}$ は状態 k に最後に到達してから状態 j に向かう経路の部分に対応する．

上に述べた (1), (2) の場合を合わせて，状態 i から j に至る経路で，状態 k より上の番号の状態を一度も通らないものの全体を表す表現として次が得られる．

$$R_{ij}^{(k)}=R_{ij}^{(k-1)}+R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^*R_{kj}^{(k-1)}$$

ここで，表現 $R_{ij}^{(k)}$ は上付き添字が $k-1$ の表現と，集合和，ドット，スター演算のみを使って構成されていることに注意せよ．よって，この式を使って上付き添字の小さ

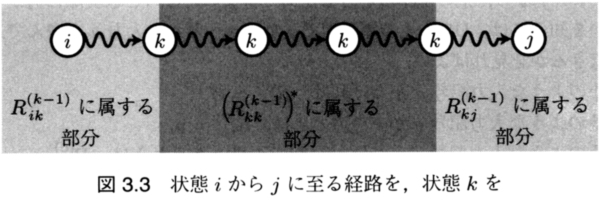


図 3.3　状態 i から j に至る経路を，状態 k を通過する点で分割した図

いものから大きいものへ順に構成することにより，すべての $R_{ij}^{(k)}$ が具体的に正則表現として（つまり，ε と $\varnothing$ と単独の記号から，集合和と連接とスター演算のみを使って）構成できることがわかる．

こうして最終的に，すべての i,j に対する $R_{ij}^{(n)}$ が求まる．このとき，A の初期状態を 1 とすると，オートマトン A の受理集合は $R_{1j}^{(n)}$ の和（ただし，j はすべての受理状態の上を動く）として与えられる． ▨

例 3.5 図 3.4 の DFA の受理集合を表す正則表現を求める．

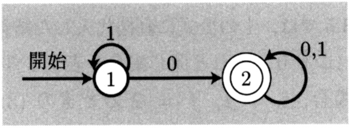


図 3.4 0 を一つ以上含む列の全体を受理する DFA

この DFA は，0 を一つ以上含む列の全体を受理する．なぜなら，このオートマトンは入力記号として 0 を読むと直ちに開始状態 1 から受理状態 2 へ移り，その後はどんな入力を読んでも状態 2 にとどまるからである．

まず最初に，定理 3.4 の構成における基礎部分の正則表現を求める．

$R_{11}^{(0)}$	$\varepsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\varnothing$
$R_{22}^{(0)}$	$(\varepsilon + \mathbf{0} + \mathbf{1})$

たとえば，$R_{11}^{(0)}$ は，始めの状態と終わりの状態が同じ 1 であるため ε を含む．また，入力記号 1 を読んで状態 1 から 1 へ向かう辺があるため $R_{11}^{(0)}$ は $\mathbf{1}$ を含む．別の例として，$R_{12}^{(0)}$ が $\mathbf{0}$ であるのは，ラベルが 0 の辺が状態 1 から 2 へ向かっているためであり，これが ε を含まないのは，始めと終わりの状態が異なるためである．さらに，$R_{21}^{(0)} = \varnothing$ であるのは，状態 2 から 1 へ向かう辺がなく，始めと終わりの状態が異なるためである．

次に，再帰的ステップで，より複雑な正則表現——まず，途中で状態 1 のみを経由し得る経路に関するもの，そして次に状態 1 と 2（すなわち，任意の状態）を経由し得る経路に関するもの——を考える．正則表現 $R_{ij}^{(1)}$ の場合は，定理 3.4 における再帰的ステップで述べた一般の場合の規則にしたがい

	直接代入した結果	簡単化した結果
$R_{11}^{(1)}$	$\varepsilon+\mathbf{1}+(\varepsilon+\mathbf{1})(\varepsilon+\mathbf{1})^*(\varepsilon+\mathbf{1})$	$\mathbf{1}^*$
$R_{12}^{(1)}$	$\mathbf{0}+(\varepsilon+\mathbf{1})(\varepsilon+\mathbf{1})^*\mathbf{0}$	$\mathbf{1}^*\mathbf{0}$
$R_{21}^{(1)}$	$\varnothing+\varnothing(\varepsilon+\mathbf{1})^*(\varepsilon+\mathbf{1})$	$\varnothing$
$R_{22}^{(1)}$	$\varepsilon+\mathbf{0}+\mathbf{1}+\varnothing(\varepsilon+\mathbf{1})^*\mathbf{0}$	$\varepsilon+\mathbf{0}+\mathbf{1}$

図 3.5　状態 1 のみを経由し得る経路を表す正則表現

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

を求めればよい．図 3.5 では，上の公式に直接代入した結果をまず示し，次にそれを簡単化した表現（つまり，それと同じ集合を表すより簡単な表現）を示す．

たとえば $R_{12}^{(1)}$ の場合，$i = 1$，$j = 2$ のときの (3.1) 式の右辺 $R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}$ に，上で得た正則表現を代入した結果と，それを簡単化した表現が示されている．簡単化の際，正則表現 R に対して一般に $(\varepsilon+R)^* = R^*$ が成り立つことを用いている．この等式が成り立つことを示すには，その両辺が共に正則表現 R の表す言語 $L(R)$ から 0 個以上任意個の要素をとりそれらを連接して得られる列全体を表すことに注意すればよい．実際，$R = \mathbf{1}$ の場合，この等式は $(\varepsilon+\mathbf{1})^* = \mathbf{1}^*$ であるが，この両辺は共に 1 のみを含む列の全体を表す．さらに $(\varepsilon+\mathbf{1})\mathbf{1}^* = \mathbf{1}^*$ でもある．なぜなら，この等式の両辺は共に 1 だけから成る列の全体を表すからである．このようにして，$R_{12}^{(1)}$ の元の表現は結局 $\mathbf{0}+\mathbf{1}^*\mathbf{0}$ と等しいことがわかる．この正則表現は（長さ 1 の）列 0 と，その前に任意個の 1 が並んだ列から成る言語を表す．しかし，その言語は実はより簡単な表現 $\mathbf{1}^*\mathbf{0}$ でも表される．

$R_{11}^{(1)}$ の簡単化も $R_{12}^{(1)}$ の場合と同様である．$R_{21}^{(1)}$ と $R_{22}^{(1)}$ の簡単化は，空集合を表す正則表現 $\varnothing$ に関する次の二つの性質を用いる．

(1)　$\varnothing R = R\varnothing = \varnothing$．これは，$\varnothing$ が連接に関する零元であること（すなわち，どんな言語に左または右から連接しても結果は常にそれ自身になるという性質を $\varnothing$ が持つことを意味する．この規則が正しいことは，上の連接結果に属する列は連接される前の二つの言語に属する列に分解されるはずだが，そのどちらかが $\varnothing$ である限りそのような列はないこと，つまり，$\varnothing$ との連接結果は常に $\varnothing$ 自身であることによる．

(2)　$\varnothing + R = R + \varnothing = R$．すなわち，$\varnothing$ は集合和に関して単位元の性質

（それとの集合和をとっても，何も変化しないという性質）を持つ．

よって，たとえば $\varnothing(\varepsilon+\mathbf{1})^*(\varepsilon+\mathbf{1})$ は $\varnothing$ に簡単化できる．このことから，その他の二つの表現の簡単化も容易にわかる．

次に，$R_{ij}^{(2)}$ の形の表現について考える．再帰的ステップで述べた公式は $k=2$ のとき

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}(R_{22}^{(1)})^* R_{2j}^{(1)} \tag{3.2}$$

である．図 3.5 の簡単化された表現を式 (3.2) に代入した結果と，それに対して上と同様の考え方に基づく簡単化を施した結果とを図 3.6 に示す．

	直接代入した結果	簡単化した結果
$R_{11}^{(2)}$	$\mathbf{1}^* + \mathbf{1}^*\mathbf{0}(\varepsilon+\mathbf{0}+\mathbf{1})^*\varnothing$	$\mathbf{1}^*$
$R_{12}^{(2)}$	$\mathbf{1}^*\mathbf{0} + \mathbf{1}^*\mathbf{0}(\varepsilon+\mathbf{0}+\mathbf{1})^*(\varepsilon+\mathbf{0}+\mathbf{1})$	$\mathbf{1}^*\mathbf{0}(\mathbf{0}+\mathbf{1})^*$
$R_{21}^{(2)}$	$\varnothing + (\varepsilon+\mathbf{0}+\mathbf{1})(\varepsilon+\mathbf{0}+\mathbf{1})^*\varnothing$	$\varnothing$
$R_{22}^{(2)}$	$\varepsilon+\mathbf{0}+\mathbf{1}+(\varepsilon+\mathbf{0}+\mathbf{1})(\varepsilon+\mathbf{0}+\mathbf{1})^*(\varepsilon+\mathbf{0}+\mathbf{1})$	$(\mathbf{0}+\mathbf{1})^*$

図 3.6 どの状態も経由し得る経路を表す正則表現

最終的に，図 3.4 のオートマトンと同値な正則表現は，このようにして得られた正則表現の中で，始めの状態が開始状態で終わりの状態が受理状態であるもの全体の和として求められる．この例の場合，1 が開始状態で 2 が唯一の受理状態であるから，図 3.6 に示したもののうちで上の条件に当てはまる正則表現は $R_{12}^{(2)} = \mathbf{1}^*\mathbf{0}(\mathbf{0}+\mathbf{1})^*$ のみである．したがって，これが求める最終結果である．この表現が表す言語は，0 個以上 1 の並んだ列で始まり，次に 0 がきて，その後に 0 または 1 がいくつでも続く列で構成される．別のいい方をすると，この言語は 0 と 1 の列で少なくとも 0 を一つ以上含むものの全体である． ▨

3.2.2 状態消去法による DFA から正則表現への変換

前節で述べた DFA から正則表現への変換はどんな DFA に対しても適用できる．さらにこの方法は，DFA だけでなく NFA や ε-NFA にも適用することができる．しかし上の方法は簡便な方法とはいえない．なぜなら，状態数が n のオートマトンの場合，約 n^3 個もの正則表現を構成しなければならないことと，再帰的ステップを n 回実行する過程でもし簡単化を行わないとすると，正則表現の長さがその度に平均で約 4 倍になるからである．

上と似た考え方を，ある種の繰り返し作業を省いて効率よく実行する方法が

ある．たとえば，定理 3.4 の構成の場合，上付き添字が (k) であるすべての表現は $(R_{kk}^{(k-1)})^*$ を部分表現として含むため，この表現を n^2 回繰り返し書く必要がある．

以下に紹介する正則表現の構成法は，オートマトンの状態を消去しながら正則表現を求める方法である．ある状態 s を消去すると，それと同時に，s を経由していた経路も消去する．そのような変更を，オートマトンの受理言語を変えずに行うには，状態 q から s を経由して状態 p へ向かう経路のラベルを，状態 q から直接 p へ至る辺のラベルに含ませるようにしなければならない．そうするためには，オートマトンの辺のラベルとして，単独の記号だけでなく記号列やその集合も認める必要がある．その際の記号列の集合は，必ずしも有限でなく，無限になることもある（元のオートマトンで s にループがある場合を考えよ）．そうすると，それらを単純に新しい辺として付け加えればよいとはいえなくなる．というのは，オートマトンの辺の個数は有限でなければならないからである．しかし，幸いそのような列のすべてを表す有限的な手段がある．正則表現を活用することである．

このようにして我々は，有限オートマトンでその遷移図のラベルが正則表現であるもののアイデアに至る．そのようなオートマトンの受理言語とは，その開始状態から受理状態に至る経路のラベルとして，その経路を構成する辺にラベル付けされた正則表現を順に連接して得られる言語全体の和として定義する．この定義の仕方は，これまで我々が見てきた各種のオートマトンに対する受理言語の定義と矛盾するものではない（たとえば，一つの記号 a または空列 ε でラベル付けされた辺は，それぞれ正則表現 **a** または ε でラベル付けされていると見ることができることに注意せよ）．

図 3.7 は，これから消去しようとしている状態 s とその周辺の状況を一般的に示す図である．ここで，状態 s の先行状態（s に向かう辺が出ている状態）が $q_1, q_2, \ldots, q_k$ であり，s の後続状態（s から出る辺の行き先）が $p_1, p_2, \ldots, p_m$ だとする．ただし，ここでは（たとい図 3.7 の場合のように s から s 自身へのループがあったとしても）s 自身を s の先行状態や後続状態とは呼ばないものとする．

さらに，状態 q_i から s に至る辺のラベルに記されている正則表現を Q_i とおき，同様に，s から p_i に至る辺のラベルに記されている正則表現を P_i とおく．

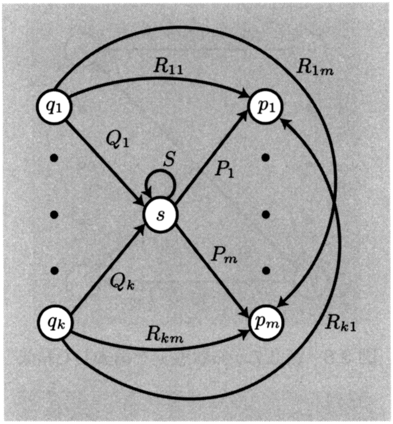


図 3.7 消去されようとしている状態 s

また，s から自分自身へ向かう辺のラベルを S とする．最後に，状態 q_i から p_j へ至る辺のラベルを正則表現 R_{ij} とする．ただし，上のどの場合においても，実際にオートマトンの辺が存在しないときは，ラベルが正則表現 $\varnothing$ の辺があると見なす．

図 3.7 の状態 s を消去した結果を図 3.8 に示す．この図では状態 s とそれに関係したすべての辺が取り除かれ，その代わりに s の各先行状態 q_i と各後続状態 p_j に対して，状態 q_i から s を経由し，そこで s を廻るループを 0 回以上廻って最後に p_j に到る経路の全体を表す正則表現 $Q_iS^*P_j$ を求め，それを状態 q_i から p_j へ向かう辺のラベル R_{ij} に（集合和の演算を用いて）加える．先に述べた通り，それまで q_i から p_j へ向かう辺がなかった場合には，R_{ij} は $\varnothing$ である．

有限オートマトンから状態消去法によって正則表現を構成する手順は次の通りである．

(1) オートマトンのすべての辺のラベルを正則表現に書き改めた後，オートマトンの各受理状態 q に対して，上の方法にしたがい q と開始状態 q_0 以外の状態をすべて消去する．

(2) $q \neq q_0$ のときは，(1) の結果として図 3.9 の形の 2 状態のオートマトンができる．このオートマトンが受理する列の全体は正則表現 $(R +$

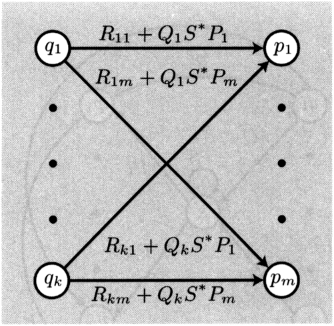


図 3.8　図 3.7 から状態 s を消去した結果

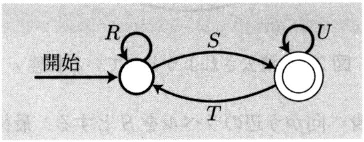


図 3.9　2 状態のオートマトンの雛型

$SU^*T)^*SU^*$ で表される．この表現の意味は次の通りである：まず，開始状態から自分自身に戻る経路として，ラベルが R の辺と SU^*T の経路がある．この後者は，まず始めにラベルが S の辺を通って受理状態に行き，次いでラベルが U の辺を何回か通ってからラベルが T の辺を通って開始状態に戻る経路を表す．開始状態から自分自身に戻るこれらの経路を何回か繰り返し通った後，ラベルが S の辺を通って最終的に開始状態を離れ，その後ラベルが U の辺を何回か通って止まる．

(3)　もし開始状態が同時に受理状態でもあるなら，元のオートマトンの開始状態以外の状態をすべて消去するためにやはり状態消去法を適用する．その結果，図 3.10 の形の 1 状態のオートマトンができる．その受理集合を表す正則表現は明らかに R^* である．

(4)　結局，最終的に求めたい正則表現は，元のオートマトンの各受理状態に対して上の (2) または (3) の方法で 2 状態または 1 状態のオートマトンの受理言語を表す正則表現を求め，それら全体の和として与えられる．

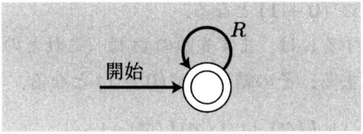


図 3.10 1 状態のオートマトンの雛型

例 3.6 図 3.11 に示した，末尾から 2 文字目か 3 文字目に 1 がある列の全体を受理する NFA について考える．我々の最初の仕事は，これを，ラベルが正則表現であるオートマトンに変換することである．そのために必要なのは，“0, 1” というラベルをそれと等価な正則表現 $\mathbf{0}+\mathbf{1}$ に書き換え，ラベル “1” を $\mathbf{1}$ に書き換えることである．その結果を図 3.12 に示す．

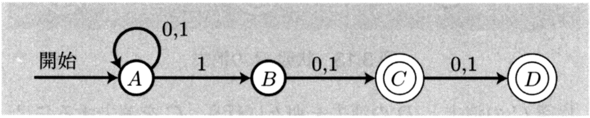


図 3.11 末尾から 2 文字目か 3 文字目に 1 がある列の全体を受理する NFA

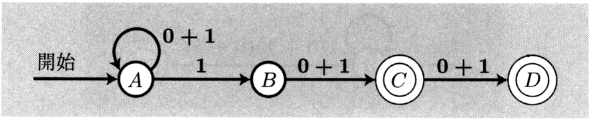


図 3.12 図 3.11 と等価で，正則表現をラベルにもつオートマトン

次に，状態 B を消去する．この状態は開始状態でも受理状態でもないため，状態を消去した後の 2 状態または 1 状態のオートマトンに残ることはない．したがって，最終的に二つの受理状態に対して二つの 2 状態オートマトンを得る過程で状態 B は必ず消去しなければならないが，それを最初に消去することにより，同じ作業を繰り返す必要がなくなる．

図 3.12 で，状態 B は A を先行状態として持ち，C を後続状態として持つ．図 3.7 における正則表現の言葉でいうと，$Q_1=\mathbf{1}$, $P_1=\mathbf{0}+\mathbf{1}$, $R_{11}=\varnothing$（状態 A から C への辺がないため），かつ $S=\varnothing$（状態 B を出て B に戻るループがないため）である．このとき，状態 B の消去により，状態 A から C へ至る

辺のラベルは $\emptyset + \mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$ となる．

この表現を簡単化するには，まず先頭の $\emptyset$ は（これとの集合和をとっても変わらないので）取り去る．その結果 $\mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$ となる．次に

$$L(\emptyset^*) = \{\varepsilon\} \ \cup \ L(\emptyset) \ \cup \ L(\emptyset)L(\emptyset) \ \cup \ \cdots$$

であり，この第 2 項目以降はいずれも空集合であるから $L(\emptyset^*) = \{\varepsilon\}$，つまり正則表現 $\emptyset^*$ は $\{\varepsilon\}$ と等価であることがわかる．よって，$\mathbf{1}\emptyset^*(\mathbf{0}+\mathbf{1})$ は $\mathbf{1}(\mathbf{0}+\mathbf{1})$ と等価である．この後者の正則表現を我々は図 3.13 で辺 $A \to C$ のラベルとして用いる．

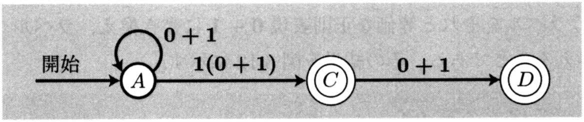


図 3.13　状態 B の消去

次に，状態 C の消去と D の消去を別々に行う．C を消去するには，先に状態 B を消去したのと同様にすればよく，その結果を図 3.14 に示す．

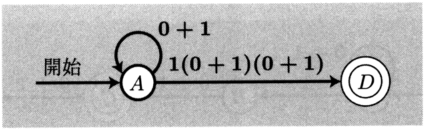


図 3.14　状態 A と D を持つ 2 状態のオートマトン

これは，図 3.9 に示した 2 状態のオートマトンの言葉でいうと，$R = \mathbf{0}+\mathbf{1}$, $S = \mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$, $T = \emptyset$，かつ $U = \emptyset$ の場合である．ここで，正則表現 U^* は ε で置き換えられるから，したがって連接の際これは消去できる．なぜなら，前に述べた通り $\emptyset^* = \varepsilon$ だからである．また，表現 SU^*T は，連接される項の一つである T が $\emptyset$ だから，全体として $\emptyset$ と等価である．したがって，正則表現 $(R+SU^*T)^*SU^*$ は今の場合，R^*S，すなわち $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$ と簡単化できる．この表現が表す言語は，1 で終わる任意の列の後に二つの文字（それぞれ 0 か 1）が付いたものの全体を意味する．この言語は図 3.11 のオートマトンの受理集合のうち，末尾から 3 文字目が 1 である列の全体である．

次に，図 3.13 のオートマトンから状態 D を消去する．D はこのオートマトンで後続の状態を持たないから，図 3.7 と図 3.8 が示す通り，この場合我々は単に C から D へ向かう辺と状態 D を削除するのみで，その他に何も変更する必要はない．そのようにして得られた 2 状態のオートマトンを図 3.15 に示す．

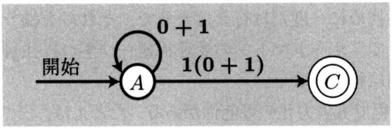


図 3.15　状態 D を消去してできる 2 状態のオートマトン

このオートマトンは図 3.14 のものとよく似ている．両者の違いは，開始状態から受理状態に向かう辺のラベルの違いだけである．したがって，我々は前と同じように 2 状態のオートマトンの受理集合を求め，その結果を簡単化することにより，$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$ を得る．この表現は図 3.11 のオートマトンの受理集合のうちの残りの部分，すなわち末尾から 2 文字目が 1 である列の全体を表す．

図 3.11 のオートマトンの受理集合に対する表現を求めるのに必要なことは，上で得た二つの表現の集合和をとることで，その結果は

$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})+(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$$

である．▨

3.2.3　正則表現からオートマトンへの変換

次に，図 3.1 に述べた計画を完結するため，どの正則表現 R についてもそれが表す言語 $L(R)$ はある ε-NFA E の受理言語 $L(E)$ であることを以下に示す．証明は，正則表現の構造に関する帰納法による．そのため，先ずはじめに帰納法の基礎の場合，すなわち，単独の記号と ε と $\emptyset$ の場合についてそれらを受理するオートマトンを構成する．その次に，これらのオートマトンを元に，それらの言語の集合和，連接，閉包をそれぞれ受理する，より大きいオートマトンを構成する方法について考える．

定理 3.7　正則表現で表される言語はすべて有限オートマトンの受理集合である．

状態を消去する順序について

例 3.6 で見たように，開始状態でも受理状態でもない状態は，状態消去法でオートマトンを 2 状態または 1 状態のものに変換する過程で必ず消去される．オートマトンの受理集合を正則表現の形で表すための方法として 3.2.1 項で述べた機械的な方法に比べて，状態消去法が持つ利点の一つは，開始状態でも受理状態でもない状態の消去は初めに一度だけ行えば十分で，それらを繰り返す必要がないことである．その上で，オートマトンの受理状態一つずつに注目して，それ以外の状態の消去を個別に行えばよい．

さらに，その過程でも省力化の可能性がある．たとえば，三つの受理状態 p, q, r が残されたとき，まず状態 p を消去し，次に q か r のどちらを消去するかによって r か q をそれぞれ受理状態とするオートマトンができる．次いで，元に戻って三つの状態 p, q, r があるところから状態 q と r を消去することにより p を受理状態とするオートマトンを求めればよい．

証明　正則表現 R に対して $L = L(R)$ のとき，$L = L(E)$ でかつ以下の条件を満たす ε-NFA E を構成する．

(1)　受理状態はただ一つである．

(2)　開始状態に向かう辺はない．

(3)　受理状態から出る辺はない．

証明は，R の構造に関する帰納法による．すなわち，3.1.2 項で述べた正則表現の再帰的な定義に基づき，R より先に構成された表現について上の条件を満たすオートマトンが存在することを仮定し，R についてもそうであることを示す．

基礎：正則表現の再帰的定義における基礎は，(a) ε，(b) $\varnothing$，(c) 単独の記号，の三つである．(a) の場合，図 3.16(a) のオートマトンの開始状態から受理状態に至る経路はただ一つで，そのラベルは ε であるから，このオートマトンの受理集合は明らかに正則表現 ε の言語である $\{\varepsilon\}$ と等しい．図 3.16(b) のオートマトンには開始状態から受理状態に至る経路がないから，このオートマトンの言語は明らかに $\varnothing$ である．最後に図の (c) は，正則表現 $\mathbf{a}$ に対するオートマトンを示す．以上の三つのオートマトンはどれも明らかに上に述べた三つの条件を満たしている．

帰納：図 3.17 に帰納的ステップの三つの場合の考え方を図解する．帰納法の原理により，今考えている正則表現の部分表現に対してはすでに上の条件を満たすオートマトンが存在すると仮定して，今考えている正則表現についてもそれが成り立つことを示せばよい．我々が考えなければならないのは次の四つの場合である．

(1)　正則表現 R と S から集合和 $R+S$ を構成した場合，$L(R)$ を受理するオートマトンと $L(S)$ を受理するオートマトンを元に，新たに開始状態と受理状態を付

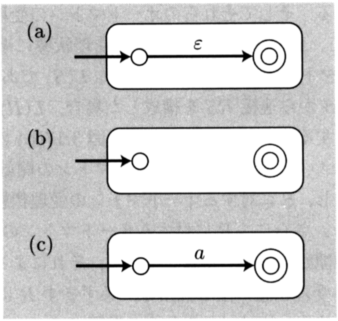


図 3.16　正則表現をもとにオートマトンを構成するための基礎

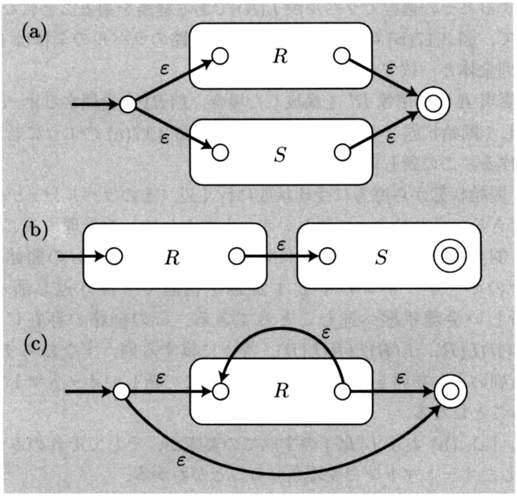


図 3.17　正則表現から ε-NFA を構成する帰納的なステップ

け加えて図 3.17(a) に示すオートマトンを作る．すると，新しい開始状態から，R に対するオートマトンの開始状態に向かうことができると同時に S に対するオートマトンの開始状態に向かうこともできる．その後，R と S のどちらかのオートマトンの中を通って各々の受理状態に到達すると，その過程を示す経路のラベル全体は，R のオートマトンの場合 $L(R)$ であり，S のオートマトンの

場合は $L(S)$ である．そしてそれらのオートマトンの受理状態を過ぎると，ラベルが ε の辺を経て，新しいオートマトンの受理状態に導かれる．こうして，図 3.17(a) のオートマトンの受理言語は $L(R) \cup L(S)$ であることがわかる．

(2)　正則表現 R と S から連接 RS を構成した場合，$L(R)$ を受理するオートマトンと $L(S)$ を受理するオートマトンを元に，図 3.17(b) のように新しいオートマトンを作る．すなわち，R に対するオートマトンの開始状態を新しいオートマトンの開始状態とし，S に対するオートマトンの受理状態を新しいオートマトンの受理状態とする．さらに，R に対するオートマトンの受理状態から S に対するオートマトンの開始状態へ ε 遷移を加える．それによって，新しいオートマトンの開始状態から受理状態へ至る経路は，必ずまず R に対するオートマトンを開始状態から受理状態に抜け（するとその過程でラベルが $L(R)$ である経路を通ることになる），次いで S に対するオートマトンを開始状態から受理状態に抜ける（するとその過程でラベルが $L(S)$ である経路を通ることになる）．このようにして，図 3.17(b) のオートマトン上の経路のラベルの全体は $L(R)L(S)$ に属する列全体と一致する．

(3)　正則表現 R から閉包 R^* を構成した場合，$L(R)$ を受理するオートマトンを元に，新しく開始状態と受理状態を付け加えて図 3.17(c) のように新しいオートマトンを作る．この新しいオートマトンでは，

　(a)　開始状態から直ちに受理状態に行く辺（そのラベルは ε）を進むことができる．それによって新しいオートマトンは ε を受理することになる．

　(b)　開始状態から元の（R に対する）オートマトンの開始状態に進み，そのオートマトンの中を 1 回以上何回でも繰り返し通って，最後に新しい受理状態へ進むこともできる．この経路の存在により，$L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$，等々に属する列，すなわち $L(R^*)$ に属する列のうち空列 ε 以外のものはすべてこの新しいオートマトンで受理されることになる．

よって，(a), (b) より $L(R^*)$ のすべての要素が，そしてそれのみが，この新しく構成したオートマトンで受理されることがわかる．

(4)　正則表現 R から (R) を構成した場合，定義により後者の表す言語は前者のそれと等しい．今，帰納法の仮定により，$L(R)$ を受理するオートマトンが存在するが，それは (R) の表す言語を受理するオートマトンでもある．

上のいずれの場合も，元になるオートマトンが最初にあげた三つの性質を満たしている限り，新しく構成したオートマトンも，それらの性質を満たしていることに注意せよ．　▨

例 3.8 正則表現 $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$ を ε-NFA に変換してみよう．そのためまず最初に，$(\mathbf{0}+\mathbf{1})$ に対するオートマトンを作る．我々は図 3.16(c) にしたがって構成した二つのオートマトン（一つはラベル 0 を持ち，もう一つはラベル 1 を持つもの）を使う．次にそれらの二つのオートマトンを図 3.17(a) の方法にしたがって組み合わせる．その結果を図 3.18(a) に示す．

次のステップでは，この図 3.18(a) のオートマトンに対して図 3.17(c) の方法を適用して $(\mathbf{0}+\mathbf{1})^*$ に対するオートマトン図 3.18(b) を作る．

最後の二つのステップでは図 3.17(b) の方法にしたがい連接に対応するオー

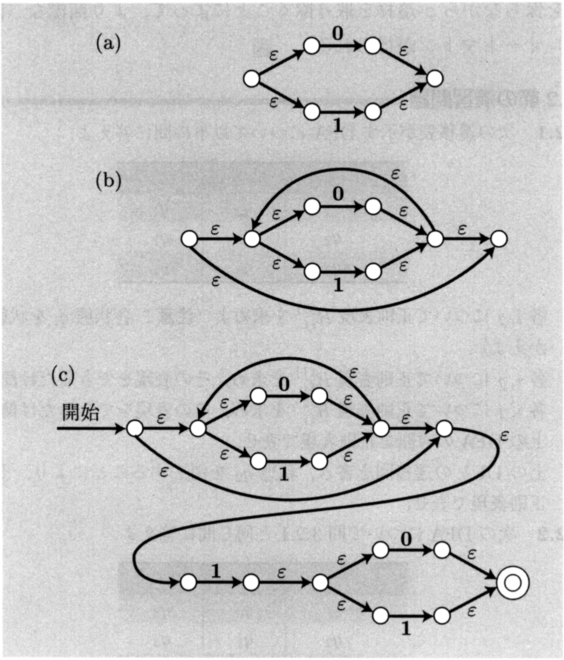


図 3.18 例 3.8 のために構成したオートマトン

トマトンの構成を 2 回行う．第 1 回目は，図 3.18(b) のオートマトンと列 1 のみを受理するオートマトン（これは図 3.16(c) でラベルを 1 としたもの）に適用する．ただし，ここで注意すべきことは，前に作って図 3.18(a) の一部分となっているものとは別に，もう一つ新しく 1 のみを受理するオートマトンを作らねばならないことである．第 2 回目は $(\mathbf{0}+\mathbf{1})$ に対するオートマトンをもう一つ（前に図 3.18(a) で作り図 3.18(b) の中に使われているものをそのまま利用するのではなく，それをコピーしたもの）を作り，それと上で作った $(\mathbf{0}+\mathbf{1})^*$ に対するオートマトンを元に，それらの言語の連接に対するオートマトンを作る．このようにして得られた最終結果を図 3.18(c) に示す．この ε-NFA から，受理言語を保ちながら ε-遷移を取り除くことによって，より簡潔な（図 3.15 と同じ形の）オートマトンが得られる． ▨

3.2.4　3.2 節の演習問題

♣　**問 3.2.1**　次の遷移表が示す DFA について以下の問に答えよ．

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

*a)　各 i, j について正則表現 $R_{ij}^{(0)}$ を求めよ（**注意：** 各状態 q_i を状態 i と読みかえよ）．

*b)　各 i, j について正則表現 $R_{ij}^{(1)}$ を求め，その表現をできるだけ簡単化せよ．

c)　各 i, j について正則表現 $R_{ij}^{(2)}$ を求め，その表現をできるだけ簡単化せよ．

d)　上の DFA の言語を正則表現で表せ．

*e)　上の DFA の遷移図を書き，状態 q_2 を消去することにより，その言語を正則表現で表せ．

♣　**問 3.2.2**　次の DFA について問 3.2.1 と同じ問に答えよ．

	0	1
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

ただし，この問の (a), (b), (e) に対する解答例は（この本のウェブ・ページに）用意されていない．

♣ **問 3.2.3**　次の DFA を，3.2.2 項で述べた状態消去法を使って正則表現に変換せよ．

	0	1
$\to *p$	s	p
q	p	s
r	r	q
s	q	r

♣ **問 3.2.4**　次の正則表現を ε-NFA に変換せよ．

*a)　$\mathbf{01}^*$.

b)　$(\mathbf{0}+\mathbf{1})\mathbf{01}$.

c)　$\mathbf{00}(\mathbf{0}+\mathbf{1})^*$.

♣ **問 3.2.5**　問 3.2.4 で構成した各 ε-NFA から ε-遷移を除去せよ．(a) の解答例がこの本のウェブページにある．

♣ **! 問 3.2.6**　$A=(Q,\Sigma,\delta,q_0,\{q_f\})$ を，q_0 に入る辺と q_f から出る辺のない ε-NFA とする．この A の言語と，A に次の修正を加えた ε-NFA の言語の間の関係を述べよ．

*a)　A に，受理状態 q_f から開始状態 q_0 に向かう ε-遷移を付け加える．

*b)　A の開始状態 q_0 から（ラベルが Σ に属する記号かまたは ε である辺をいくつか通って）到達可能なすべての状態に対して，q_0 からその状態に向かう ε-遷移を付け加える．

c)　A において，受理状態 q_f に到達可能なすべての状態から q_f に向かう ε-遷移を付け加える．

d)　A に上の (b) と (c) の変更を加える．

♣ **!! 問 3.2.7**　正則表現から ε-NFA を構成する定理 3.7 の方法に関していくつかの簡略化が可能である．たとえば，

(1)　集合和に関して，新たに開始状態と受理状態を設ける代わりに，元の二つのオートマトンの開始状態を一つの状態にまとめてもよい（それに伴い今までどちらかの開始状態から出ていた遷移をすべてそのまとめた状態から出すようにする）．同じように，二つのオートマトンの受理状態を一つの状態にまとめてもよい．

(2)　連接に関して，最初のオートマトンの受理状態と 2 番目のオートマトンの開始状態を一つにまとめてもよい．

(3)　閉包演算に関して，単に，受理状態から開始状態に向けて ε-遷移を新たに加え，逆に開始状態から受理状態に向けて ε-遷移を加えてもよい．

上に述べたそれぞれの簡略化はそれ自身正しい結果をもたらす．すなわち，上で述べた方法をどんな正則表現に適用しても，その表現の言語を受理する ε-NFA が得られる．ところで，上の (1), (2), (3) の変更を組み合わせるとき，どのような組合せ方なら一般の正則表現に対して正しいオートマトンが構成できるか?

♣ !! **問 3.2.8** DFA A と整数 n が与えられたとき，A によって受理される長さ n の列がいくつあるかを計算するアルゴリズムを示せ．ただし，そのアルゴリズムは，A の状態数と n に関して多項式時間で計算するものとする（**ヒント：**定理 3.4 の構成法を利用せよ）．

3.3 正則表現の応用

正則表現の応用

正則表現は，我々が文字列のパターンを認識しようとするときの「視覚的イメージ」に近い概念であり，そのため，テキスト中のパターン検索などに応用できる可能性がある．その場合，正則表現は舞台裏で決定性または非決定性のオートマトンに変換され，その動きにしたがってテキスト中のパターン（に該当する文字列）を認識するプログラムが作られる．この節で我々は，正則表現に基づく二つの重要な応用として，字句解析とテキスト検索の場合を紹介する．

3.3.1 UNIX における正則表現

応用について述べる前に，UNIX で用いられる正則表現の拡張を紹介する．その拡張された記法は，いくつかの新しい機能を提供するが，その中にはたとえば，パターンに適合した文字列があったらそれに名前を付け，後でそれを参照する機能などのように，それによって正則でない言語の認識が可能になるものもある．しかしここではその種の拡張には触れず，複雑な正則表現を簡潔に記述するための略記法についてのみ述べる．

最初に紹介する一群の拡張は，UNIX における正則表現の使用のほとんどが ASCII 文字を扱うことに由来するものである．本書のこれまでの例では，たとえば $\{0, 1\}$ のような小さなアルファベット（文字集合）を用いていたが，その場合「任意の文字」を表すには単純に $\mathbf{0} + \mathbf{1}$ と書けばよかった．しかし，たとえばアルファベットが 128 文字から成るとき，それらの文字をすべて列挙するのは現実的でない．そのため，UNIX の正則表現ではある種の文字の集合を簡潔に記述するための「文字クラス」と呼ばれる表現が用意されている．文字ク

ラスは以下の規則にしたがって構成する．

- 記号．（ドット）で任意の 1 文字を表す．
- $[a_1 a_2 \cdots a_k]$ で，次の正則表現を表す．

$$\mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$$

  この記法は，演算記号 + を省略することにより実際にタイプする文字数を約半分にする効果がある．たとえば，C 言語で数値などを比較するために使われる四つの記号の集合は [<>!=] で表される．
- 大括弧の中に x-y と書くと，ASCII の文字表における x から y までのすべての文字を指定したことになる．ただしこの表の中では，数字，大文字の英字，小文字の英字は，それぞれ通常の順序にしたがって並べられている．そのためこの記法を用いることにより，我々が実際に入力する必要のある文字クラスの多くが少数のキーを叩けば打ち込めるようになっている．たとえば，数字は [0-9] で，英字の大文字は [A-Z] でそれぞれ表され，英数字全体は [A-Za-z0-9] で表される．さらに，上のリストに負の記号を加えたい場合，それをリストの最初か最後に付け加えることにより，文字の範囲を示す記法との混同が避けられる．たとえば，符号付き十進数を表すために使う，数字と小数点と正負の記号から成る文字クラスは [-+.0-9] で表せる．UNIX の正則表現で特殊な目的のために使われる記号（たとえば大括弧など）は，その直前にバックスラッシュ（\）を書くことにより他の文字と同様に扱うことができる．
- 頻繁に使われるいくつかの文字クラスについては，特別の記法が用意されている．たとえば，
  a)　[:digit:] は数字のクラス [0-9] を表す[4]．
  b)　[:alpha:] は英字のクラス [A-Za-z] を表す．
  c)　[:alnum:] は英数字のクラス [A-Za-z0-9] を表す．

以上の他に，UNIX の正則表現で使われる演算記号でこれまでの話に登場しなかったものがいくつかある．これらの演算記号を使うことによって本質的に表現力が増大する訳ではないが，書きたいことがより容易に書ける場合がある．

---

[4] この記法は，たとえば ASCII 以外の文字表で，特に数字が文字表の中で連続して並んでいないものを使うときなどに役立つ．

(1) 集合和を表すのに，+ の代わりに演算記号 | を使う．

(2) 演算記号 ? は「0回または1回の」を表す．すなわち，UNIX で $R?$ と書くと本書の $\varepsilon + R$ を意味する．

(3) 演算記号 + は「1回以上の繰り返し」を意味する．すなわち，UNIX の $R+$ は RR^* の略記である．

(4) 演算記号 $\{n\}$ は「n 回の繰り返し」を意味する．すなわち，UNIX での $R\{5\}$ は $RRRRR$ の略記である．

3.1.2 項で，正則表現の部分表現を括る括弧と，演算の優先順位について述べたが，その点についてはUNIX の正則表現も同じ規則にしたがう．ただし，?, +, $\{n\}$ の演算の優先順位は * と同じである．* 演算は UNIX の正則表現でも我々の場合と同じ意味で（ただし，もちろん上付き文字としてでなく）用いる．

3.3.2 字句解析

最も古くから始まった正則表現の応用の一つに,「字句解析プログラム（lexical analyzer）」と呼ばれる，コンパイラの一部での利用があげられる．字句解析プログラムはコンパイラの中で，ソースプログラムを読みながら，その中に現れるトークンの認識を行う部分を指す．ただしトークンとは，プログラム中の連続した文字列で，プログラムの論理的な最小単位の役割を果たすものであり，たとえば，プログラミング言語に特有なキーワードや，種々の名前（identifier）や，演算記号など，多くの種類がある．

UNIX コマンド `lex` とその GNU 版 `flex` は，トークンの具体的な形をUNIX 流の正則表現で記述したものと，そのトークンを認識したとき字句解析プログラムがとるべきアクション（のコード）の組を，認識すべきすべてのトークンについて列挙したリストを入力として受け取る．これらのコマンドは，この

UNIX の正則表現の詳細

UNIX の正則表現で使える演算記号や略記法の全体を知りたい読者は，UNIX のいろいろなコマンドのマニュアルページを見よ．それらは UNIX の版によって若干相違があるが，たとえばコマンド `man grep` を実行すると，基本的なコマンドである `grep` で使用可能な記法が見られる．なお，grep とは Global search for Regular Expression and Print の略である．

ように字句解析の際に必要となる情報を一般的な形で記述したものを入力データとして読み，それをもとに字句解析プログラムを自動的に作成するという意味で，字句解析プログラム生成系（lexical-analyzer generator）と呼ばれる．

コマンド `lex` や `flex` は，正則表現がプログラミング言語のトークンの形を記述するのに必要な表現力を持っているため非常によく利用されている．これらのコマンドでは，ソースプログラムをトークンに分解するための効率的なプログラムが，正則表現を DFA に変換することにより自動生成される．こうしたコマンドの誕生により，字句解析プログラムの作成は半日程度の簡単な仕事となったが，それ以前は，手作業でそれを書くのに数ヶ月を要することもあった．また，一旦完成した字句解析プログラムを変更する際も，以前は複雑なコードの中に分け入ってバグを訂正するなどの苦労があったが，今では正則表現を少し書き換えれば済む．

例 3.9　`lex` コマンドの入力例の一部を紹介する意味で，図 3.19 に C 言語のいくつかのトークンに関する記述例を示す．

最初の行は，C のキーワード `else` に関するもので，それに対するアクションは，字句解析の次に実行される構文解析プログラムにそのトークンの種類を示す定数（ここでは `ELSE`）を渡すことである．2 番目の行では，(変数や関数な

```
else                    {return(ELSE);}

[A-Za-z][A-Za-z0-9]*    {code to enter the found identifier
                         in the symbol table;
                         return(ID);
                        }

>=                      {return(GE);}

=                       {return(EQ);}

...
```

図 3.19　`lex` の入力例

どにつける）名前の形を示す正則表現として，最初が英字でその後に0個以上の英数字が続く列を指定し，これに対するアクションとして，その名前をコンパイラの記号表に（もしまだ入っていなければ）登録することと，そのトークンが名前であることを示すための定数 ID を構文解析プログラムに渡すことが記されている．

図3.19の3行目は，2文字からなる演算記号 >= に関するものであり，最後の例は1文字の演算記号 = に関するものである．実際には，プログラム中に現れるすべてのキーワードや，符号，句読点，括弧，種々の定数など，すべてのトークンに関する記述をこのような形で与える．これらのうちの多くはただ一つの記号列から成る単純なものだが，たとえば上に示した名前と同程度に複雑なものもあり，それらを記述するのに正則表現が有用である．そのようなトークンの例として他に，整数や，浮動小数点数，文字列，コメント，などがある．
▨

図3.19に例示したような正則表現の集まりをオートマトンに変換する作業は，大筋において前節までに述べた方法で行われる．ただし，各トークンごとに正則表現を変換するのではなく，それらの集合和に対するオートマトンを一つ生成する．すると，このオートマトンから得られる情報は「どれかの」トークンが認識されたということだが，しかし，定理3.7の集合和の場合の構成法を見ればわかるように，ε-NFA の状態からどのトークンが認識されたかを読み取ることができる．

ただ一つ問題となり得るのは，一度に二つ以上のトークンが認識されるかも知れないという点である．たとえば，文字列 else は正則表現 else に適合する他に，名前を表す正則表現にもマッチする．この問題に対する標準的な解決法は，字句解析プログラム生成系が最初に出会う正則表現に優先権を与えるというものである．したがって，もし else のようなキーワードを予約語（プログラマが名前として使うことができない文字列）にしたければ，else に対する記述を名前に対するものより前におけばよい．

3.3.3 テキスト中のパターン検索

2.4.1 項で，ウェブのような大量の情報庫を効率よく検索して，指定した（有限個の）単語を探し出すのにオートマトンが使えることを述べた．そのための道具や技術は字句解析の場合ほどよく整備されているとはいえないが，正則表現はパターン検索の際に役立つ様々のパターンを記述するのに有用である．一方，字句解析の場合は，正則表現がトークンに対する自然な記述言語であると同時に，それを元に（オートマトンに基づく）効率的なプログラムが自動的に作成されるということが，有用な知的手段を提供する結果となったといえる．

ところで，パターン検索の場合，どんな種類の問題を解決するのに正則表現の技術がこれまで有用だったかを考えてみると，曖昧な形で認識されているパターンをテキストの中から探そうという場合がそれに当たるといえよう．その記述が曖昧だということは，そのパターンを初めから正確に記述するのは困難だということを意味する――もしかすると正確な記述をそれに対して与えるのは永久に無理かも知れない．そのような場合に，正則表現を用いることによって，そのパターンをまず大まかな形であまり苦労せずに記述し，その後もし不都合が生じたら訂正を加えていくことが比較的楽にできる．その際，正則表現を読んで，そのパターンに基づく検索を実際に行うプログラムを作り出す「コンパイラ」があれば有用である．

ウェブの様々な応用において現実に起こりそうなその種の問題を例にとってこのことを具体的に考えてみよう．たとえば今我々は，大量のウェブページを調べてその中から住所を抽出したいとする．それは，単に新しいメーリングリストを作るためであったり，あるいは，職業を地区別に分類して，「自分が今居るところから車で 10 分以内のレストランを見付けてほしい」というような要求に答えるためかも知れない．

そのような目的のため，特に（英語圏における） street address を認識する問題を考えよう．ところで，そもそも street address とは何だろうか? それを先ず明らかにしなければならないが，そのために取りあえず自分の考えたものを試してみて，その上で，欠けているものに気付いたらそれを取り込むように表現を書き直すことにする．

まず最初は，street address として “Street” またはその省略形である “St.”

で終わるものを考えることにしよう．ただし，場合によっては “Avenue” や “Road”，あるいはそれらの短縮形が使われることもある．このようにして取りあえず我々の正則表現の最後部の候補として次を考える．

```
Street|St\.|Avenue|Ave\.|Road|Rd\.
```

上の表現で我々は，集合和を表すのに + ではなく | を用いる UNIX の記法にしたがっている．また，UNIX の正則表現ではドットが「任意の文字」という特別の意味を持つが，我々が書きたいのはそれではなく「ドット」の記号そのもののため，ドットの前にバックスラッシュを書く．

“Street” などの表示の前に，通りの名前を記す必要がある．通りの名前は普通，英字の大文字で始まりその後に英字の小文字がいくつか続いた形である．そのようなパターンは UNIX の正則表現 `[A-Z][a-z]*` で表される．しかし，Washington DC における Rhode Island Avenue のように，2 語から成る通りの名前もあることを考えると，通りの名前の記述を次のように変更する必要がある．

```
'[A-Z][a-z]*( [A-Z][a-z]*)*'
```

この表現は，大文字の後に小文字がいくつか続いた形で始まり，その後に空白で始まり次いで大文字がありその後にいくつかの小文字が続いた形が 0 回以上繰り返された列の全体を表す．空白は UNIX の表現では通常の文字として扱えるが，UNIX のコマンドライン上でそれが二つの表現（を空白を挟んで並べたもの）と見なされることを避けるため，この表現全体を引用符で括る必要がある．この引用符は，正則表現の一部とは見なされない．

次に，住所の一部として家の番地を上の表現に加える．普通，番地は数字の列である．しかし，たとえば “123A Main St.” のように数字列の後に大文字の英字が一つ付くこともある．そのことを考慮すると，結局，番地の表現は `[0-9]+[A-Z]?` となる．ここで，+ は「1 個以上の」数字の列を表すために用いられた UNIX の演算であり，? は「0 個または 1 個の」大文字を表すための UNIX の演算である．このようにして street address に対する正則表現として，結局，次が得られた．

```
'[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*
```

```
    (Street|St\.|Avenue|Ave\.|Road|Rd\.)'
```

この表現を実際に試してみるとかなり満足のいく結果が得られそうである．しかし，やがてたとえば以下の点が考慮されていないことに気付くであろう．

(1) 通りには street, avenue, road 以外の名前が付いていることもある．たとえば，“Boulevard”, “Place”, “Way” やそれらの簡略形など．

(2) 数や，あるいは “42nd Street” のように一部が数である通りの名前もある．

(3) 私書箱（Post-Office box）や地方配達路（rural-delivery route）の場合もあり得る．

(4) Street などの語で終わらない通りの名前もある．たとえば，Silicon Valley における El Camino Real は，スペイン語で royal road を意味し，El Camino Real road とはいわない．したがって，2000 El Camino Real のような住所の扱い方も考える必要がある．

(5) 我々の想像を越えるその他の風変わりな名前もある．

このようにして，もし正則表現を認識するコンパイラがあれば，住所を完全に認識するプログラムに次第に近づいて行くことができる．そのようなコンパイラなしに自分で直接プログラムを書きそれに修正を加えて行くのと比べると，前者の方が仕事ははるかに容易だと思われる．

3.3.4 3.3 節の演習問題

♣ !! **問 3.3.1** 考えられるあらゆる形の電話番号を記述する正則表現を示せ．国際電話と，それぞれの国によって局番や地域の番号の長さが異なることとを考慮せよ．

♣ !! **問 3.3.2** 求人広告に現れる給料を示す正則表現を求めよ．給料には，時給，週給，月給，年俸があることを考慮せよ．ドルの表示があるものとないもの，“K” などの単位が後に付くもの，なども考慮せよ．また，それが給料であることを示す語が脇に書き添えられているものもある．新聞の 3 行広告を見てどんなパターンが相応しいか検討することを勧める．

♣ ! **問 3.3.3** 3.3.3 項の終わりに，住所を表す正則表現に対して考えられる改善を例示した．先に示した正則表現にこれらの改善をすべて加えよ．

3.4 正則表現の代数的法則

例3.5で，正則表現のサイズが大きくなり過ぎるのを抑えるため，正則表現を簡単化する必要が生じた．その際，ある正則表現が他の正則表現で置き換えられる理由についてアドホックな議論を行った．その議論で肝心なことは，その二つの正則表現が同じ言語を表すという意味で等価であるという点である．この節では，一連の代数的な法則を示し，その後で，どのような場合に二つの正則表現が等価かという問題をより高い視点から見直す．すなわち，個々の具体的な正則表現を調べる代わりに，変数（またはパラメタ）を含む正則表現を取り上げる．そのような変数を含んだ二つの正則表現が**等価**（equivalent）であるとは，それらの変数にどんな言語を代入しても，二つの表現から同じ言語が得られることをいう．

こうした変数を含む表現への代入というプロセスについて，算術の場合を例にとって説明する．$1+2=2+1$ というのは具体的な二つの式の値が等しいことを述べているが，これは，足し算の交換法則と呼ばれる一般的な法則の一つの具体例である．この具体的な式が正しいことは，両辺の値を求めると $3=3$ と成ることからわかる．しかし，足し算の交換法則はもっと多くのこと，つまり $x+y=y+x$ における変数 x と y にそれぞれどんな数を代入しても両辺の値は等しいことを述べている．すなわち，どんな二つの数を加えても，それらを加える順序に関係なく同じ答が得られることをこの法則は意味する．

算術の式の場合と同様に，正則表現の場合もそれらの間に成り立ついくつかの法則がある．その多くは，集合和をとる演算を算術の場合の足し算と見なし，連接演算を算術の掛け算と見なしたとき，算術の法則と似ている．しかし，算術との類似性が保たれないことや，正則表現に関して成り立つ法則で算術の場合に対応するものがないものもある（特に閉包演算に関する法則の場合にそれが当てはまる）．以下に，正則表現に関する等式が正しい法則であるか否か（すなわち，変数にどんな言語を代入してもその等式が成り立つか否か）を調べる方法について論じる．

3.4.1 結合法則と交換法則

交換法則は演算に関する性質で，演算対象の順序を変更しても結果は変わらないという性質をいう．算術について先に述べた $x+y=y+x$ はその例であ

る．また，結合法則も演算に関する性質で，演算対象をいくつかのグループに分けてその演算を適用して最終的に一つの結果を出す場合，そのグループ分けの仕方が結果に影響しないことをいう．たとえば，$(x \times y) \times z = x \times (y \times z)$ は掛け算の結合法則である．正則表現について成り立つこの種の法則は次の三つである．

- $L + M = M + L$. これは**集合和に関する交換法則**（commutative law for union）で，二つの言語の集合和をとるとき，それらの言語の順序は問題にならないことを述べている．
- $(L+M)+N = L+(M+N)$. これは**集合和に関する結合法則**（associative law for union）で，三つの言語の集合和をとるとき，最初の二つの言語の和を先にとっても後の二つの言語の和を先にとっても結果は変わらないことを述べている．上の和に関する交換法則と併せて考えると，要するに，いくつかの言語の集合和を求めるとき，どんな順序やグループ分けを行って集合和を求めても結果は同じであることがわかる．直観的にいうとそれは，ある列が $L_1 \cup L_2 \cup \cdots \cup L_k$ に属するのは，その列が一つ以上の L_i に属するとき，かつそのときに限るためである．
- $(LM)N = L(MN)$. これは**連接に関する結合法則**（associative law for concatenation）で，三つの言語を連接する際，最初の二つを先に連接しても，後の二つを先に連接しても結果は同じだということを述べている．

上のリストに欠けているのは連接に関する交換 “法則” $LM = ML$ で，実際それは正しくない．

例 3.10　正則表現 **01** と **10** について考える．これらはそれぞれ言語 $\{01\}$ と $\{10\}$ を表し，この両者は明らかに等しくない．このことから，$LM = ML$ は正しい法則でないことがわかる．なぜなら，この式の L に正則表現 **0** を代入し M に **1** を代入すると，左辺は **01** で，右辺は **10** となるが，この両者は上で述べた通り等しくないからである．　▨

3.4.2　単位元と零元

ある演算の**単位元**（identity）とは，その演算を単位元と他の値，たとえば x に適用すると結果は x のままであることをいう．たとえば，$0 + x = x + 0 = x$ であるから，0 は足し算の単位元である．また，$1 \times x = x \times 1 = x$ であるか

ら，1は掛け算の単位元である．一方，ある演算の**零元**（annihilator）とは，その演算を零元と他の値，たとえばxに適用すると結果は零元であることをいう．たとえば，$0 \times x = x \times 0 = 0$であるから，0は掛け算の零元である．しかし，足し算に零元はない．

正則表現の場合，この種の概念に関する法則として次の三つがある．

- $\varnothing + L = L + \varnothing = L$. この法則は$\varnothing$が集合和に関する単位元であることを意味する．
- $\varepsilon L = L\varepsilon = L$. この法則は$\varepsilon$が連接に関する単位元であることを意味する．
- $\varnothing L = L\varnothing = \varnothing$. この法則は$\varnothing$が連接に関する零元であることを意味する．

これらの法則は正則表現の簡単化において効力を発揮する．たとえば，いくつかの表現の集合和を求める際，そのうちのあるものがもともと$\varnothing$であるか，あるいは簡略化した結果$\varnothing$に成るなら，それは和をとる対象から取り去ってよい．同様に，いくつかの表現の連接を求める際，そのうちのあるものがもともとεであるか，あるいは簡単化した結果εになるなら，それは連接をとる対象から取り去ってよい．最後に，いくつかの表現の連接を求める際，そのうちの一つでも$\varnothing$なら，結果も$\varnothing$である．

3.4.3 分配法則

分配法則は二つの演算に関する法則であり，第1の演算を第2の演算の結果に対して適用するのと，第2の演算を行う前に前もってその各演算対象に個別に第1の演算を適用するのとでは，同じ結果に成ることを意味する．算術の場合によく知られた例は，足し算に対する掛け算の分配法則$x \times (y + z) = x \times y + x \times z$である．ここで，掛け算について交換法則が成り立つから，掛け算を和の左から行っても右から行っても結果は変わらない．正則表現の場合も同様の法則が成り立つが，連接について交換法則が成り立たないため，左からの連接と右からの連接を別々に述べる必要がある．すなわち，

- $L(M + N) = LM + LN$. この法則は**集合和に対する連接の左分配法則**（left distributive law of concatenation over union）と呼ばれる．
- $(M + N)L = ML + NL$. この法則は**集合和に対する連接の右分配法則**（right distributive law of concatenation over union）と呼ばれる．

上の二つの分配法則のうち，左分配法則が成り立つことを次の定理で示す．右分配法則についても同様である．

定理 3.11 任意の言語 L, M, N について次が成り立つ．

$$L(M+N) = LM + LN.$$

証明 証明は，定理 1.10 で述べた分配法則の場合と同様に次のことを示せばよい：ある列 w が $L(M+N)$ に属するためには，それが $LM+LN$ に属することが必要かつ十分な条件である．

（必要性）もし w が $L(M+N)$ に属するならば，$w = xy$ と書ける．ただしここで x は L に属し y は M または N に属する．もし y が M に属するならば，xy は LM に属し，したがって $LM + LN$ にも属する．同様に，もし y が N に属するならば，xy は LN に属し，したがって $LM + LN$ にも属する．

（十分性）w が $LM + LN$ に属するとする．このとき，w は LM または LN に属する．w が LM に属するならば，$w = xy$ の形に書ける．ただし x は L に属し y は M に属する．今 y は M に属するから，$M + N$ にも属する．したがって $w = xy$ は $L(M+N)$ に属する．w が LM に属さないときは LN に属し，上と同様の議論によりやはり $L(M+N)$ に属することがいえる． ▨

例 3.12 正則表現 $\mathbf{0} + \mathbf{01}^*$ の簡単化について考えよう．まず，最初の $\mathbf{0}$ の部分に注目すると，$\mathbf{0}$ は $\mathbf{0}$ と ε の連接である．言い替えると，連接に関する単位元の法則より $\mathbf{0} = \mathbf{0}\varepsilon$ が成り立つ．したがって，与式は $\mathbf{0}\varepsilon + \mathbf{01}^*$ と書き換えられる．この式はさらに左分配法則により $\mathbf{0}(\varepsilon + \mathbf{1}^*)$ と書き換えられる．ここで，ε は言語 $L(\mathbf{1}^*)$ に属するから，$\varepsilon + \mathbf{1}^* = \mathbf{1}^*$，したがって与式は $\mathbf{01}^*$ と簡単化できる． ▨

3.4.4 冪 等 法 則

ある（2 変数の）演算が**冪等**（idempotent）であるとは，同じ値にその演算を適用すると常に元の値が得られることをいう．算術の足し算や掛け算の場合，一般に $x + x = x$ は成り立たない（$0 + 0 = 0$ のように成り立つ場合もあるが，すべての場合に成り立つ訳ではない）し，$x \times x = x$ も一般には成り立たないから，これらの演算は冪等ではない．しかし，集合和と共通部分をとる演算はそれぞれ冪等演算である．したがって，正則表現について次の法則が成り立つ．

- $L+L=L$. これは**集合和に関する冪等法則** (idempotence law for union) で，二つの同じ言語の集合和は元の言語と等しいことを意味する．

3.4.5 閉包に関する法則

閉包演算およびその変型と見なせる UNIX の + 演算や ? 演算に関する法則を以下に列挙し，それらが正しい理由を簡単に述べる．

- $(L^*)^* = L^*$. この法則は，言語の閉包を 2 回とっても 1 回の場合と変わらないことを主張する．ここで左辺 $(L^*)^*$ は，L^* に属する列を任意個取り，それらを連接して得られる列から成る．しかし，L^* に属する列はそれ自身 L から任意個の要素を取りそれらを連接した結果である．したがって，$(L^*)^*$ の要素も L から任意個の要素をとりそれらを連接した結果であり，それらは L^* に属する．逆に L^* の要素が $(L^*)^*$ に属することは閉包演算の定義から明らかである．
- $\varnothing^* = \varepsilon$. 空集合 $\varnothing$ の閉包は ε のみを含むことは例 3.6 で述べた通りである．
- $\varepsilon^* = \varepsilon$. 空列 ε をいくつ連接しても得られる結果は常に空列である．
- $L^+ = LL^* = L^*L$. L^+ は定義により $L+LL+LLL+\cdots$ である．一方，$L^* = \varepsilon + L + LL + LLL + \cdots$ であるから，したがって

$$LL^* = L\varepsilon + LL + LLL + LLLL + \cdots$$

  ここで $L\varepsilon = L$ であることを使うと，LL^* の無限展開と L^+ に対するそれが一致することがわかる．よって $L^+ = LL^*$ が成り立つ．$L^+ = L^*L$ の証明も同様である[5]．
- $L^* = L^+ + \varepsilon$. この証明は，L^* と L^+ の無限展開を考えれば容易である．実際，L^+ の展開は L^* の展開の最初の ε を除いた部分に等しい．なお，L に ε が入っている場合は，この法則の右辺の「$+\varepsilon$」の部分は不要で，$L^* = L^+$ が成り立つ．
- $L? = \varepsilon + L$. これは法則というより演算 ? の定義である．

---

[5] このことから，任意の言語 L はその閉包 L^* と連接において交換可能であることがわかる．すなわち，$LL^* = L^*L$ が成り立つ．このことは，一般に連接について交換法則は成り立たないという事実と矛盾するものではない．

3.4.6 正則表現に関する法則の発見

上に述べた法則はすべて証明されているが，その他にも正則表現に関して成り立つかも知れない法則の候補は数限りなくある．それらが果たして正しい法則かどうかを簡単に調べる一般的な方法はあるだろうか？

法則は，正則表現に関する沢山の等式を変数を用いてまとめて表したものだが，そのような法則の正しさを示す問題は，ある具体的な一つの等式の正しさを調べる問題に還元することができる．興味深いことに，その技法は正則表現で用いられる演算と密接な関係があり，たとえば共通部分などの演算を含む法則の場合にその技法を拡張することはできない．

この技法がどのようなものかを説明するため，法則の候補として次のものを取り上げる．

$$(L+M)^* = (L^*M^*)^*$$

この法則は，どんな言語 L, M についても，両者の集合和の閉包は L^*M^*（L から任意個要素をとって並べた後に M から任意個要素をとって並べてできる列の全体）の閉包に等しいことを述べている．

この法則を証明するには，まず，左辺 $(L+M)^*$ の言語[6]から任意の要素 w をとる．すると，ある k と $L+M$ に属する列 $w_1, w_2, \cdots, w_k$ を用いて $w = w_1w_2\cdots w_k$ と書ける．このとき，各 w_i は L^*M^* の要素でもある．なぜなら，もし w_i が L の要素なら，それは $L^* = \{\varepsilon\} \cup L \cup L^2 \cup \cdots$ の要素である．そして L^* の要素は $L^*M^* = L^*(\{\varepsilon\} \cup M \cup M^2 \cup \cdots) = L^* \cup L^*M \cup L^*M^2 \cup \cdots$ の要素でもあるからである．w_i が M の要素の場合も同様である．このようにしてすべての w_i が L^*M^* に属するから，結局 $w = w_1w_2\cdots w_k$ はその閉包である $(L^*M^*)^*$ に属する．

上記の法則の証明を完結するには，上の逆，つまり $(L^*M^*)^*$ の要素は $(L+M)^*$ の要素でもあることも示さなければならないが，ここではそれは行わない．その理由は，今我々がすべきことは，この法則を証明することではなく，正則表現に関する重要な次の性質を指摘することだからである．

---

6) 簡単のため，正則表現とその言語とを同一視して,「の言語」という言葉を以後省略する．<**訳注**：厳密に言うと，この $(L+M)^*$ は L, M を変数として含む正則表現で，その言語 $L((L+M)^*)$ とは $(L+M)^*$ の変数 L, M に任意の言語を代入して得られる結果を意味する．したがって「$(L+M)^*$ の言語」は必ずしも正則言語ではない．>

変数と ε と $\varnothing$ に正則表現の三つの演算を適用して得られる表現（それを「変数を含む正則表現」と呼ぶ）に対して，その中に現れる各変数を一つの特殊な記号と見なすことにより，それ自身を一つの具体的な（変数を含まない）正則表現と見ることができる．たとえば，表現 $(L+M)^*$ において，変数 L, M をそれぞれ特殊な記号とみるか，あるいはより明示的に，新しい記号たとえば a, b で変数 L, M をそれぞれ置き換えることにより，具体的な正則表現 $(\mathbf{a}+\mathbf{b})^*$ が得られる．

このようにして得られた具体的な正則表現 $(\mathbf{a}+\mathbf{b})^*$ の言語は，その元になった，変数を含む正則表現 $(L+M)^*$ の言語（すなわち，$(L+M)^*$ の変数 L, M に任意の言語を代入した結果）の要素がどのように構成されるか，その骨格を示すものといえる．上で $(L+M)^*$ の分析を行った際，我々は $(L+M)^*$ の要素が L か M に属する列を任意個とって並べた結果であることを使ったが，その事実は，具体的な正則表現 $(\mathbf{a}+\mathbf{b})^*$ の要素（a と b から成る任意の列）によって端的に示されている．つまり，この正則表現 $(\mathbf{a}+\mathbf{b})^*$ の要素中の a, b にそれぞれ言語 L, M の任意の要素を代入した結果を全部集めたものが $(L+M)^*$ の要素となる．ただしここでの代入は，a （または b）の異なる出現に対して L（または M）の異なる要素を代入してもよい．

上で述べたことは自明に思われるかも知れないが，後の囲み記事「正則表現以外の表現に上の判定法を拡張した場合の失敗例」で指摘するように，正則表現の三つの演算以外のものを含む表現の場合には成り立つとは限らない．次の定理で，変数を含む正則表現について一般的に成り立つある性質を証明する．

定理 3.13 E を，変数 $L_1, L_2, \ldots, L_m$ を含む正則表現とする．この E の中の各 L_j を記号 a_j で置き換えてできる具体的な正則表現を C とする．すると，E の中の変数 $L_1, L_2, \ldots, L_m$ をそれぞれ具体的なある言語と見たとき正則表現 E の表す言語 $L(E)$ は，$L(C)$ の要素，たとえば $a_{j_1}a_{j_2}\cdots a_{j_k}$ からその中の各記号 a_{j_i} を対応する言語 L_{j_i} の任意の要素で置き換えることによって得られる列全体に等しい．

証明 変数を含む正則表現 E の構成に関する帰納法による．

基礎：E が ε か $\varnothing$ か変数の場合を考えればよい．そのうち，はじめの二つの場合は具体的な正則表現 C は元の表現 E と同じため，何も証明することはない．E が変数

L の場合，$L(E) = L$ である．一方，具体的な正則表現 C は，変数 L に対応する記号を a とすると，$\mathbf{a}$ であり，$L(C) = \{a\}$. ここで $L(C)$ の唯一の要素 a に言語 L の任意の要素を代入した結果の全体は明らかに L，すなわち $L(E)$ である．

帰納：E を構成する際，最後に適用した演算に応じて，三つの場合に分けて考える．まず $E = F + G$ の場合を考える．今，F と G の中の変数を記号に置き換えて得られる具体的な表現をそれぞれ C と D とする．ただし，その際 F と G の中の同じ変数は同じ記号で置き換えるものとする．そのとき，E から得られる具体的な正則表現は $C + D$ である．次に，E 中の変数をそれぞれ具体的な言語と見たとき，$L(E)$ の要素 w は $L(F)$ の要素かまたは $L(G)$ の要素である．もし w が $L(F)$ の要素なら，帰納法の仮定から，w は $L(C)$ のある要素の各記号に，対応する言語の要素をそれぞれ代入することにより得られる．このとき，$L(C + D) = L(C) \cup L(D)$ より，w は $L(C + D)$ のある要素の各記号に，対応する言語の要素をそれぞれ代入することにより得られる．w が $L(G)$ の要素の場合も同様である[7]．正則表現 E が FG または F^* の形の場合も上と同様の考え方で証明できるが，詳細は読者に委ねる．▨

3.4.7　正則表現に関する法則の検証

以下に，正則表現に関する法則の正しさを知るための判定法を述べ，その判定法が正しいことを定理 3.14 で示す．E, F が共に変数を含む正則表現のとき，等式 $E = F$ が正しい法則であるか否かを判定する方法は次の通りである．

(1) E と F の中に現れる各変数を対応する記号で置き換えることにより具体的な正則表現 C と D をそれぞれ得る．

(2) $L(C) = L(D)$ か否かを調べる．もしそうなら $E = F$ は正しい法則であり，そうでなければその「法則」は正しくない（二つの具体的な正則表現が同じ言語を表すか否かを判定する一般的な方法については後の 4.4 節で述べる．ただし，実際に与えられた二つの正則表現が等しいことを示すにはアドホックな方法が使えるかも知れない．二つの言語が等しくないことを示すには，反例（一方の言語に入っていて他方に入っていない列）を一つ見付ければよい）．

---

7) <**訳注**：以上で，$L(E)$ の各要素が $L(C + D)$ の要素から，各記号 a_j を対応する言語 L_j の要素で置き換えて得られることが示された．逆に $L(C + D) = L(C) \cup L(D)$ の要素からそのようにして得られた列は，帰納法の仮定から $L(F) \cup L(D)$ に属するから $L(E)$ に属することが言える．>

定理 3.14 上の判定法により，正則表現に関する等式が正しい法則か否かが調べられる．

証明 変数を含む正則表現 E, F に対して，$E = F$ が正しい法則である（つまり，その中の変数に任意の言語を代入したとき $L(E) = L(F)$ が成り立つ）ために $L(C) = L(D)$ が必要かつ十分な条件であることを示せばよい．

（必要性）E, F の中の変数に任意の言語を代入したとき $L(E) = L(F)$ が成り立つとする．そのとき特に，各変数 L に代入する言語として，表現 C, D を得る際に代入した記号 a のみから成る言語 $\{a\}$ を選ぶ．するとこのとき，$L(C) = L(E)$，かつ $L(D) = L(F)$ であるから，仮定 $L(E) = L(F)$ より $L(C) = L(D)$ を得る．

（十分性）$L(C) = L(D)$ とする．定理 3.13 により，$L(E)$ と $L(F)$ は，それぞれ $L(C)$ と $L(D)$ の具体的な記号をそれに対応する言語の要素で置き換えて得られる列全体に等しい．しかるに，仮定 $L(C) = L(D)$ より，上の方法で得られる二つの言語 $L(E)$ と $L(F)$ は明らかに一致する．すなわち $L(E) = L(F)$ が成り立つ． ▨

例 3.15 $(L + M)^* = (L^*M^*)^*$ が法則か否かを上の定理の方法で判定する．そのため，変数 L, M をそれぞれ具体的な記号 a, b で置き換えると，上の等式の両辺は $(\mathbf{a} + \mathbf{b})^*$ と $(\mathbf{a}^*\mathbf{b}^*)^*$ となる．この二つの表現は共に a と b から成る列全体を表すから，両者は等しい．よって定理 3.14 より $(L + M)^* = (L^*M^*)^*$ は正しい法則である．

もう一つの法則の候補として $L^* = L^*L^*$ を考える．この両辺の変数 L を具体的な記号 a に書き換えた結果はそれぞれ $\mathbf{a}^*$ と $\mathbf{a}^*\mathbf{a}^*$ であり，これらは共に a から成る列全体を表すから，ゆえに $L^* = L^*L^*$ は正しい法則である．

最後の例として，$L + ML = (L + M)L$ を取り上げる．ここで両辺の変数 L を a に，変数 M を b にそれぞれ書き換えると，正則表現 $\mathbf{a} + \mathbf{ba}$ と $(\mathbf{a} + \mathbf{b})\mathbf{a}$ が得られる．しかし，これらの言語は等しくない．なぜなら，たとえば aa は $(\mathbf{a} + \mathbf{b})\mathbf{a}$ に入るが $\mathbf{a} + \mathbf{ba}$ には入らない．よって $L + ML = (L + M)L$ は正しい法則ではない． ▨

3.4.8 3.4 節の演習問題

♣ **問 3.4.1** 正則表現の形で表される次の法則を証明せよ．

*a) $R + S = S + R$.

b) $(R + S) + T = R + (S + T)$.

c) $(RS)T = R(ST)$.

d)　$R(S+T) = RS + RT$.

e)　$(R+S)T = RT + ST$.

*f)　$(R^*)^* = R^*$.

g)　$(\varepsilon + R)^* = R^*$.

h)　$(R^*S^*)^* = (R+S)^*$.

♣ **! 問 3.4.2**　正則表現の形で表される次の等式のうち，正しい法則を証明し，正しくないものについては反例を示せ．

*a)　$(R+S)^* = R^* + S^*$.

b)　$(RS+R)^*R = R(SR+R)^*$.

*c)　$(RS+R)^*RS = (RR^*S)^*$.

d)　$(R+S)^*S = (R^*S)^*$.

e)　$S(RS+S)^*R = RR^*S(RR^*S)^*$.

♣ **問 3.4.3**　例 3.6 で我々は次の正則表現を得た．

$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1}) + (\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$$

分配法則を用いて，この表現と等価でより簡単な二つの正則表現を求めよ．

♣ **問 3.4.4**　3.4.6 項の始めに，$(L^*M^*)^* = (L+M)^*$ の証明を途中まで述べた．$(L^*M^*)^*$ に入る列は $(L+M)^*$ にも入ることを示してこの証明を完結せよ．

♣ **! 問 3.4.5**　定理 3.13 の証明を，正則表現 E が FG または F^* の形の場合を示すことにより完結せよ．

正則表現以外の表現に上の判定法を拡張した場合の失敗例

正則表現に共通部分の演算を加えた表現を考える．興味深いことに，正則表現の三つの演算に共通部分の演算 $\cap$ を加えても，表現できる言語の範囲は変わらない（このことは後の定理 4.8 で述べる）．

今 $L \cap M \cap N = L \cap M$ という“法則”について考える．これは，三つの言語の共通部分は最初の二つの共通部分に等しいという主張である．この“法則”は，たとえば $L = M = \{a\}$ かつ $N = \varnothing$ の場合を考えると，明らかに誤りである．しかし，“法則”中の変数を具体化するという上に述べた判定法はこの場合，両辺の違いを示すのに役立たない．実際，変数 L, M, N をそれぞれ記号 a, b, c に置き換えたとき，この式は $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$ と成るが，両辺とも空集合であり，この式自身は正しい．よって，もし先の判定法がこの場合にも拡張できるとすると，この“法則”は正しいことになる．

3.5 第3章の要約

◆**正則表現**：この代数的な記法は，有限オートマトンと同じく正則言語を，そしてそれのみを記述することができる．正則表現で用いる演算は，集合和，連接（ドット），閉包（スター）の三つである．

◆**正則表現の応用**：UNIX のようなオペレーティングシステムのいろいろなコマンドで，ある種の略記法を含む正則表現が用いられている．この拡張された正則表現では，記号の集まりに対して文字クラスと呼ばれる便利な省略形が提供され，また,「1個以上の」とか「0 個または 1 個の」のような演算が用意されている．

◆**正則表現と有限オートマトンの等価性**：DFA を正則表現に変換する方法として，DFA の遷移図上の経路についての記述を，その経路の途中で通過できる頂点の範囲を次第に拡げていくというやり方で，簡単なものから複雑なものへ再帰的に構成していく方法がある．またそれとは別に，状態消去法によって DFA に対する正則表現を求めることもできる．逆の正則表現から DFA を求める方向は，正則表現からまず ε-NFA を求め，それを DFA に変換する方法がある．

◆**正則表現の代数的性質**：正則表現は，算術における代数的性質と類似の（ただし，同じではない）沢山の法則にしたがう．集合和と連接は結合法則を満たすが，交換法則は集合和だけが満たす．連接は集合和に対して分配的であり，集合和は冪等的である．

◆**恒等式の判定法**：変数を含む二つの正則表現間の等式が恒等式か否かを判定する問題は，変数を記号で置き換えることによってできる具体的な（変数を含まない）正則表現の等価性を調べる問題に還元することができる．

3.6 第3章の参考文献

正則表現の考え方と，それが有限オートマトンと等価であることの証明は，S.C.Kleene [3] による．しかし，本章で述べた，正則表現から ε-NFA を作る方法は [4] に基づき,「McNaughton-Yamada の構成法」と呼ばれる．正則表現の等式を，その中に現れる変数を定数として判定する方法は J.Gisher [2] により発表された．この考え方自体はその前から広く知られていたと思われるが，上の報告書は，正則表現の構成に使用される以外の，たとえば共通部分やシャッフル（問 7.3.4 を見よ）などの演算を加えると,（それらの演算を加えても言語の表現力は前と同じだが）この判定法は使えなくなることを示している．

UNIX が作られる前から，K.Thompson は正則表現を `grep` などのコマンドで使うことを検討していた．それらのコマンドを実行するための彼のアルゴリズムが [5] に発表されている．早期の UNIX では，たとえばレスク（M.Lesk）による `lex` コマ

ンドのように，正則表現の拡張版を多用したコマンドがいくつか存在した．この `lex` コマンドと，正則表現を使ったその他の技法についての解説が [1] にある．

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

[2] J. L. Gischer, STAN-CS-TR-84-1033(1984).

[3] S. C. Kleene, “Representation of events in nerve nets and finite automata,” In C. E. Shannon and J.McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp.3-42.

[4] R. McNaughton and H.Yamada, “Regular expressions and state graphs for automata,” *IEEE Trans. Electronic Computers* **9**:1 (Jan., 1960), pp.39-47.

[5] K. Thompson, “Regular expression search algorithm,” *Comm. ACM* **11**:6 (June, 1968), pp.419-422.

邦 訳 文 献

[1] 原田賢一訳：コンパイラ I, II—原理・技法・ツール—（サイエンス社，1990）

4 正則言語の性質

本章では正則言語の性質についてさらに深く探求を進める．この探求のために我々が用いる最初の道具は，ある特定の言語が正則ではないことを証明するための一つの手法である．「反復補題」と呼ばれるこの定理は，4.1 節で紹介される．

正則言語に関する一つの重要な性質に「閉包性」と呼ばれる類のものがある．これらの性質により，ある言語からある種の演算によって得られる別の言語に対し，その言語を認識する機械の構成が可能になる．たとえば，二つの正則言語の共通部分は，やはり正則言語である．相異なる二つの正則言語に対し各々を認識する二つのオートマトンが与えられたとき，その二つの言語の共通部分を認識するオートマトンを，機械的に構成することができる．ただし，共通部分を認識するオートマトンは，最初に与えられた二つのオートマトンのいずれよりもずっと多くの個数の状態を持つかもしれない．この種の「閉包性」を複雑なオートマトンを構成するための有用な道具として利用することもできる．4.2 節では，このような構成を本質的に用いて議論を進めていく．

正則言語に関する別の重要な性質に「決定可能性 (decision property)」と呼ばれるものがある．この種の性質の研究を通して，オートマトンに関する重要な質問に答えるためのアルゴリズムが与えられる．中心的な例は，二つのオートマトンが同一の言語を定義するか否かを判定するアルゴリズムである．この問題が判定可能であることの一つの帰結として，我々は，オートマトンを「最小化」すること，すなわち，与えられたオートマトンと等価なオートマトンで状態数がもっとも少ないものを見出すことができる．この問題は，スイッチング回路の設計において数十年もの間とても重要な問題であった．回路の費用（回路が占めるチップの面積）は，回路によって実現されるオートマトンの状態数が減少するのにつれて減少するからである．

4.1 言語が正則でないことの証明

正則言語と呼ばれる言語のクラスには少なくとも四つの異なる記述法があることを見てきた．それらはDFA，NFA，および，ε-NFAの各々によって受理される言語としての記述であり，また，正則表現によって定義される言語としての記述である．

すべての言語が正則言語であるというわけではない．本節では，ある言語が正則ではないことを示すための一つの強力な手法を紹介する．これは「反復補題」という名で知られている．次に，正則ではない言語の例をいくつか示す．さらに，4.2節では，反復補題を正則言語の閉包性と組み合わせて，いわば，2頭立てで，言語が正則でないことを証明する方法を見ていく．

4.1.1 正則言語に対する反復補題

言語 $L_{01} = \{0^n1^n \mid n \geq 1\}$ を考えよう．この言語は，01, 0011, 000111 のように，前半が1個以上の0の列であり，後半がそれと同数個の1の列であるようなすべての文字列から成る．L_{01} は正則言語ではないことを示そう．証明の論法を直観的に述べると次のようになる．もし，L_{01} が正則であるとすると，L_{01} はある DFA A によって受理される．このオートマトンはある一定の状態数を持つ．A の状態数を k としよう．このオートマトンが入力として k 個の0から成る列を受け取った状況を考えてみる．A は，入力列の $k+1$ 個の接頭語，$\varepsilon, 0, 00, \ldots, 0^k$，の各々を読み終わるごとにある状態をとる．相異なる状態は k 個しかないので，「鳩の巣原理 (pigeonhole principle)」により，二つの相異なる接頭語，たとえば 0^i と 0^j $(i \neq j)$ を読み終わった後，A は同じ状態，たとえば q をとることになる．

ところで，オートマトン A が i 個または j 個の0を読んだ後で，さらに，いくつかの1の列を入力として受け取るとする．受け取る1の個数が i 個の場合，もし A がその前に i 個の0を受け取っていたとすれば A は入力列を受理しなければならず，一方，j 個の0を受け取っていたとすれば A は入力列を受理してはならない．1を読み始める時点で状態 q にあったとすると，A は，それ以前に i 個の0を受け取っていたのか，それとも，j 個の0を受け取っていたのかを「思い出す」ことができない．そこで，我々は，A を「だまし」て，受理すべきでないときに受理させたり，受理すべきときに受理しないように，誤っ

た動作をさせてしまうことができる．

上に記した説明は十分に厳密なものではないが，厳密な証明に書き直すことは容易である．さて一方，言語 L_{01} は正則ではないという同じ結論は，次に示すもっと一般的な結果を用いて導くことも可能である．

定理 4.1 **（正則言語に対する反復補題）** L を正則言語とする．(L に依存する）定数 n で，L に属し，かつ，$|w| \geq n$ を満たす任意の文字列 w に対し，w を次の条件を満たす 3 個の部分列の連接 $w = xyz$ に分解できるようなものが存在する．

(1)　$y \neq \varepsilon$.

(2)　$|xy| \leq n$.

(3)　すべての $k \geq 0$ に対し，文字列 xy^kz もまた L に属する．

いいかえると，w の先頭からあまり遠くない位置にある空でない文字列 y で，「ポンプで汲み上げる」ことが可能なものが存在する．(反復補題はポンプの補題とも呼ばれる.）すなわち，y を好きな回数だけ繰り返し連接する，または y を消去する（$k = 0$ の場合）という操作を施してもその結果得られる文字列は必ずまた言語 L に属するという性質を持つ y を常に見つけることができる．

証明　L は正則言語であるとする．そうすると，ある DFA A で $L = L(A)$ を満たすものが存在する．A は n 個の状態を持つと仮定する．いま，長さ n 以上の任意の文字列 w を考える．$w = a_1a_2\cdots a_m$ とする．ただし，$m \geq n$ であり，a_i はいずれも一つの入力記号である．$i = 0, 1, \ldots, n$ の各々に対し，状態 p_i を $p_i = \hat{\delta}(q_0, a_1a_2\cdots a_i)$ と定める．ここで，δ は A の状態遷移関数であり，q_0 は A の初期状態である．すなわち，p_i は A が w の先頭の i 個の記号を読み終えた後にとる状態である．$p_0 = q_0$ であることに注意する．

異なる状態は全部で n 個しかないので，鳩の巣原理により，$n+1$ 個の p_i ($i = 0, 1, \ldots, n$) がすべて相異なるということはあり得ない．したがって，二つの異なる整数 i と j ($0 \leq i < j \leq n$) で $p_i = p_j$ を満たすものが存在する．さて，そこで，文字列 w を次を満たすように $w = xyz$ と分解する．

(1)　$x = a_1a_2\cdots a_i$

(2)　$y = a_{i+1}a_{i+2}\cdots a_j$

(3)　$z = a_{j+1}a_{j+2}\cdots a_m$

すなわち，x により p_i に到達し，y により p_i から再び p_i に戻る（p_i と p_j は等しいことに注意）．z は w の残りの部分である．文字列と状態との間の関係を図 4.1 に

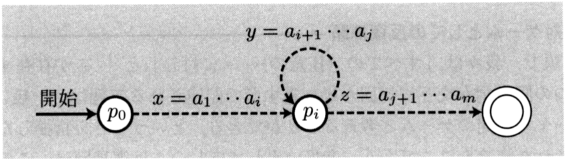


図 4.1　状態数より長い任意の文字列は必ず状態の重複を引き起こす．

示す．$i = 0$ の場合もあり得るので，x は空列であるかもしれないことに注意されたい．また，$j = n = m$ ということもあり得るので，z も空列である可能性がある．しかし，y は空列ではあり得ない．i は j より真に小さいからである．

さて，オートマトン A が，任意の $k \geq 0$ に対し，入力 xy^kz を受け取るとき，どんなことが起こるかを考えてみよう．まず，$k = 0$ の場合，オートマトン A は入力 x により初期状態 q_0（これは p_0 に等しい）から状態 p_i に進む．p_i は p_j に等しいので，図 4.1 に示されているように，A は入力 z により p_i から受理状態に到達することとなる．したがって，A は xz を受理する．

次に，$k > 0$ の場合，A は入力 x により q_0 から状態 p_i に進み，入力 y^k により p_i から p_i までを k 回循環し，さらに，入力 z により受理状態に到達する．

よって，すべての $k \geq 0$ に対し，xy^kz は A によって受理される．すなわち，xy^kz は L に属す．　▨

4.1.2　反復補題の応用

反復補題がどのように使われるか，いくつかの例について見てみよう．いずれの場合も，ある特定の言語に対し，その言語が正則言語ではないことを証明するために反復補題が利用される．

例 4.2　同じ個数の 0 と 1 を含む（ただし，並び方の順序は問わない）文字列の全体から成る言語を L_{eq} とおく．L_{eq} は正則言語ではないことを証明しよう．「敵対ゲームとしての反復補題」という表題の枠内で説明されている「2人ゲーム」の言葉を用いて証明を進めることにする．我々をプレーヤー 1 とし，プレーヤー 1 はプレーヤー 2 が選ぶどのような選択に対しても適切に対処していかなければならないものとする．L_{eq} を正則言語と仮定するとき，反復補題によって存在が保証されている定数を n とする．すなわち，“プレーヤー 2” は n を選ぶ．それに対し，我々は，確実に L_{eq} に属する 一つの文字列として $w = 0^n1^n$，すなわち，列の前半に n 個の 0 があり，後半に n 個の 1 が続く列

敵対ゲームとしての反復補題

1.2.3 項で，我々は,「すべての（任意の）$\cdots$ に対し」と「$\cdots$ が存在する」という二つの限定詞が交互に何回か繰り返す形で記述される定理は，一般に，2 人のプレーヤーの間のゲームと考えることができる，ということを指摘した．そこで展開された議論をここでもう一度思い出してほしい．反復補題は，このような型の定理の重要な例の一つである．実際，反復補題には四つの異なる限定詞が含まれている.「**任意の**正則言語 L に対し，ある n が**存在し**，L に属し $|w| \geq n$ を満たす**任意の**文字列 w に対し，... であり，$w = xyz$ となる x, y, z が**存在する.**」さて，反復補題を，ゲームとして再構成してみよう．プレーヤー 1 は L が正則ではないことを主張しようとし，プレーヤー 2 は L が正則であることを主張しようとする．

(1) プレーヤー 1 は，正則ではないことを証明したい言語 L を選ぶ．

(2) プレーヤー 2 は n を選ぶ．ただし，プレーヤー 1 に対して n がいくつであるかを明かすことはしない．プレーヤー 1 はすべての可能な n に対して適用できるようプレイを行わなければならない．

(3) プレーヤー 1 は w を選ぶ．w は n に依存し，長さは n 以上でなければならない．

(4) プレーヤー 2 は w を x, y, z に分割する．その分割は，反復補題に示されている条件にしたがう．すなわち，$y \neq \varepsilon$ および $|xy| \leq n$ を満たす．ここでも，プレーヤー 2 はプレーヤー 1 に x, y, z について，反復補題の条件を満たすことを除き，それらが何であるかを伝える必要はない．

(5) プレーヤー 1 は，n, x, y, z に依存する整数 k で，xy^kz が L に属さないようなものを選べば “勝つ”．

を選ぶ．

さて，“プレーヤー 2” は，我々の w を xyz に分割する．我々が知っているすべては，$y \neq \varepsilon$ と $|xy| \leq n$ だけである．しかしながら，この情報は極めて有用であり，我々は次のようにしてゲームに “勝つ” ことができる．$|xy| \leq n$ であり，また，xy は w の先頭部分にあることから x と y は 0 だけからなる列であることがわかる．L_{eq} が正則であるならば，反復補題より xz は L_{eq} に属す．この結論は，反復補題の $k = 0$ の場合にあたる[1]．さて，w の中の 1 はすべて

[1] 以下の議論は，$k = 2$ を選んだとしても，さらには 1 以外の任意の自然数を k として選んだ場合にも，同様にうまくいくことに注意されたい．

z に含まれるので，xz は n 個の 1 を含む．一方，xz は n 個より少ない個数の 0 しか含まない．y に含まれる 0 は xz からは失われているからである．$y \neq \varepsilon$ であるので，x と z の中には 0 は $n-1$ 個以下しか存在し得ない．こうして，我々は，L_{eq} は正則言語であると仮定したのち，xz は L_{eq} に属す，という正しくない命題を証明することができた．L_{eq} は正則ではないという事実の背理法による証明が得られたことになる．▨

例 4.3　1 だけから成る列で長さが素数であるものの全体から成る言語を L_{pr} とおく．L_{pr} は正則言語ではないことを証明しよう．L_{pr} は正則言語であると仮定する．そうすると，反復補題の条件を満たす定数 n が存在する．p を $p \geq n+2$ を満たす素数とする．素数は無限個あるので，必ずそのような p は存在する．$w = 1^p$ とする．

反復補題によれば，w を，$y \neq \varepsilon$ および $|xy| \leq n$ を満たす x, y, z に対し，$w = xyz$ と分解することができる．$|y| = m$ とおく．したがって $|xz| = p - m$ である．さて，文字列 $xy^{p-m}z$ を考えよう．L_{pr} が本当に正則であるとすれば，反復補題により $xy^{p-m}z$ は L_{pr} に属さなければならない．ところが，次式が成り立つが容易にわかる．

$$
\begin{aligned}
|xy^{p-m}z| &= |xz| + (p-m)|y| = p - m + (p-m)m \\
&= (m+1)(p-m)
\end{aligned}
$$

さて，このことから，$|xy^{p-m}z|$ は二つの因子 $m+1$ と $p-m$ を持つので素数ではない，と直ちに結論づけられそうに思えるが，しかしその前に，これらの因子はどちらも 1 ではないことを確認する必要がある．そのことを確認しないと，$(m+1)(p-m)$ は素数である可能性が残るからである．まず，$y \neq \varepsilon$ より $m \geq 1$ であるので $m+1 > 1$ であることがわかる．また，p は $p \geq n+2$ を満たすように選ばれていることと，

$$
m = |y| \leq |xy| \leq n
$$

より，$p - m \geq 2$ であることが導かれる．

ここでも前の例と同様に，我々は，問題の言語は正則であると仮定することから始めて，実際はその言語に属さないある文字列が，反復補題によるとその言語に属さなければならなくなる，ということを示して，矛盾を導いた．こうして，L_{pr} は正則言語ではないことが証明された．▨

4.1.3 4.1 節の演習問題

♣ **問 4.1.1** 次に示す言語はいずれも正則言語ではないことを証明せよ.

a) $\{0^n 1^n \mid n \geq 1\}$. この言語は，前半がいくつかの 0 の列で，後半がそれと同数個の 1 の列であるようなすべての文字列から成る言語であり，我々が本節の冒頭で直観的な方法で考察した言語 L_{01} である．ここでは，反復補題を適用して証明していただきたい.

b) 正しい順序で並べられた左括弧と右括弧の列の全体から成る集合．これは，たとえば，正しい算術式から，そこに現れる記号 “(” と “)” を抜き出して得られる記号列の集合である.

*c) $\{0^n 10^n \mid n \geq 1\}$.

d) $\{0^n 1^m 2^n \mid n$ と m は任意の自然数 $\}$.

e) $\{0^n 1^m \mid n \leq m\}$.

f) $\{0^n 1^{2n} \mid n \geq 1\}$.

♣ **! 問 4.1.2** 次に示す言語はいずれも正則言語ではないことを証明せよ.

*a) $\{0^n \mid n$ は完全平方数 $\}$.

b) $\{0^n \mid n$ は完全立方数 $\}$.

c) $\{0^n \mid n$ は 2 のベキ乗（すなわち, $n = 2^k$, $k \geq 0$）$\}$.

d) 0 と 1 とから成る文字列で，長さが完全平方数であるものの全体から成る集合.

e) 0 と 1 とから成る文字列で，ww の形のもの，すなわち，ある文字列が 2 回繰り返した形のものの全体から成る集合.

f) 0 と 1 とから成る文字列で，ww^R の形のもの，すなわち，ある文字列とその文字列を逆順に並べたものを連接して得られる文字列の全体から成る集合.

g) 0 と 1 とから成る文字列で，$w\overline{w}$ の形のものの全体から成る集合．ただし，$\overline{w}$ は，w において 0 を 1 に置き換え，1 を 0 に置き換えて得られる列である．たとえば，$\overline{011} = 100$ である．また，011100 はこの言語に属する文字列の一例である.

h) $w1^n$ の形の文字列の全体から成る集合．ただし，w は 0 と 1 とから成る任意の文字列であり，n は w の長さである.

♣ **!! 問 4.1.3** 次に示す言語はいずれも正則言語ではないことを証明せよ.

a) 0 と 1 とから成る文字列で，先頭の文字は 1 であり，2 進整数として解釈したとき素数になるものの全体から成る集合.

b) 最大公約数が 1 であるような i と j に対し，$0^i 1^j$ の形で表される文字列の全体から成る集合.

♣ !問 4.1.4　次の正則言語に対して反復補題を適用しようとすると，"敵側" が "勝つ" ことになってしまい，我々は証明を完成させることができない．L として次の各々の言語を選ぶとき，証明はどこでうまくいかなくなるのかを示せ．

*a)　空言語

*b)　$\{00, 11\}$.

*c)　$(\mathbf{00} + \mathbf{11})^*$.

d)　$\mathbf{01^*0^*1}$.

4.2　正則言語に関する閉包性

本節では,「正則ないくつかの言語に対し，言語 L はそれらの言語にある種の演算を施すことによって得られる（例：L は二つの正則言語の和である）ならば，L もまた正則である」という形の定理をいくつか証明する．これらの定理で述べる性質は，しばしば，正則言語の**閉包性** (closure property) と呼ばれる．正則言語のクラスはそこで用いられる演算のもとで閉じているということを示す性質だからである．閉包性は，一つ（または複数個）の言語が正則であるとき，それらに関連して定まるある種の言語もまた正則である，という構想を表現するものである．また，正則言語には互いに同値ないくつかの表現（オートマトンや正則表現）があるが，閉包性の証明ではある一つの表現が他の表現よりもはるかに有効に機能するという現象をしばしば経験する．したがって，閉包性は，言語のクラスに対する我々の理解において，いくつかの同値な表現がいかに互いを強化し合うかを示す興味深い例ともなっている．正則言語に対する閉包性の主要なものを以下にまとめておく．

(1)　二つの正則言語の和は正則である．

(2)　二つの正則言語の共通部分は正則である．

(3)　正則言語の補集合は正則である．

(4)　二つの正則言語の差は正則である．

(5)　正則言語の反転は正則である．

(6)　正則言語の閉包（スター閉包）は正則である．

(7)　二つの正則言語の連接は正則である．

(8)　正則言語の準同型（文字への文字列の代入）の像は正則である．

(9)　正則言語の逆準同型の像は正則である．

4.2.1 ブール演算のもとでの正則言語の閉包性

最初に示す閉包性は 三つのブール演算，すなわち，和，共通部分，補集合である．

(1) L と M をアルファベット Σ 上の言語とする．このとき，$L \cup M$ は，L と M の一方，または，両方に属する文字列の全体から成る言語である．

(2) L と M をアルファベット Σ 上の言語とする．このとき，$L \cap M$ は，L と M の両方に属する文字列の全体から成る言語である．

(3) L をアルファベット Σ 上の言語とする．このとき，L の**補集合** $\overline{L}$ は，Σ^* に属する文字列のうち，L に属さない文字列の全体から成る言語である．

正則言語のクラスは，これらの三つの演算のすべてのもとで閉じていることが示される．ただし，これから見ていくように，それらの証明にはそれぞれ異なる手法が用いられる．

和のもとでの閉包

定理 4.4 L と M が正則言語であるとき，$L \cup M$ もまた正則言語である．

証明 証明は簡単である．L と M は正則であるので，これらの言語には正則表現が対応する．たとえば，$L = L(R)$, $M = L(S)$ としよう．このとき，正則表現に対する演算子 $+$ の定義より $L \cup M = L(R + S)$ である． ▨

補集合のもとでの閉包 和に関する定理は，言語の正則表現による表示を利用することで極めて容易に証明することができた．次に，補集合について考えてみよう．読者の皆さんは，正則表現が与えられたとき，それを補集合に対応する正則表現に変形していく直接的な方法がおわかりだろうか．実は，私たちにもその方法はよくわからない．でも，そのような正則表現が存在することはわかる．定理 4.5 に示すように，DFA から出発すれば，それが受理する言語の補集合を受理する DFA を容易に構成することができるからである．したがって，正則表現から出発して，それが表す言語の補集合に対応する正則表現を求めるには，次のようにすればよい．

(1) 正則表現を ε-NFA に変換する．

(2) 得られた ε-NFA を部分集合構成法により DFA に変換する．

(3) 得られた DFA の受理状態と非受理状態を交換する．

正則演算のもとでの閉包

正則言語のクラスが和のもとで閉じていることの証明は，例外的といってよいほど容易であった．それは，和の演算が正則表現を定義する三つの演算のうちの一つだったからである．定理 4.4 の証明で用いられた論法は，連接や閉包に対しても同様に適用することが可能である．すなわち，以下が成り立つ．

- L と M がともに正則言語であるとき，LM も正則言語である．
- L が正則言語であるとき，L^* も正則言語である．

(4) 受理状態と非受理状態を交換した DFA を，3.2.1 項と 3.2.2 項に記した構成法を用いて，正則表現に戻す．

定理 4.5 L がアルファベット Σ 上の正則言語であるとき，$\overline{L} = \Sigma^* - L$ もまた正則言語である．

証明 ある DFA $A = (Q, \Sigma, \delta, q_0, F)$ に対し，$L = L(A)$ であるとする．このとき，DFA $B = (Q, \Sigma, \delta, q_0, Q - F)$ に対し，$\overline{L} = L(B)$ である．B は，A の受理状態が B の非受理状態であり，A の非受理状態が B の受理状態であることを除き，A と全く同じである．容易にわかるように，w が $L(B)$ に属すことは $\hat{\delta}(q_0, w)$ が $Q - F$ に属すことと同値であり，さらに，後者は w が $L(A)$ に属さないことと同値である． ▨

上記の証明において，$\hat{\delta}(q_0, w)$ は常にある一つの状態として確定すること，すなわち，A において遷移先が存在しないような状況は起こり得ないことが，重要なポイントであることに注意されたい．仮に，遷移先が存在しないことが起こり得るとすれば，ある文字列に対し A は受理状態にも非受理状態にも到達しないことがあり得る．このとき，それらの文字列は $L(A)$ からも $L(B)$ からも抜け落ちてしまうことになる．しかし，幸いなことに，我々は，Σ の任意の文字に対し任意の状態からの遷移が存在するように DFA を定義した．したがって，すべての文字列が，F に属す状態か，または，$Q - F$ に属す状態に必ず到達するのである．

例 4.6 図 2.14 に示したオートマトンを A とする．DFA A は，0 と 1 とから成る文字列で末尾が 01 であるものを，そして，それらだけを受理するオートマトンであった．正則表現で表すと，$L(A) = L((\mathbf{0} + \mathbf{1})^*\mathbf{01})$ となる．$L(A)$ の補集合は，0 と 1 とから成る文字列で末尾が 01 ではないものの全体である．

言語が異なるアルファベット上で定義されていたらどうするか?

二つの言語 L と M の和や共通部分を考えようとするとき，それらが相異なるアルファベット上で定義された言語であることもあり得る．たとえば，$L_1 \subseteq \{a,b\}^*$ であるが，一方，$L_2 \subseteq \{b,c,d\}^*$ であるというような場合である．しかし，言語 L が Σ 上の文字列から成る言語であるとき，同時に，L は Σ を含む任意のアルファベット上の言語であるとみなすことが可能である．したがって，上の例のような場合，L_1 と L_2 はいずれもアルファベット $\{a,b,c,d\}$ 上の言語であると考えることができる．L_1 に属す文字列の中に c や d を含むものがないことや，L_2 に属す文字列の中に a を含むものがないことなどは，重要なことではない．

同様に，あるアルファベット Σ_1 に対し，Σ_1^* の部分集合である言語 L の補集合を考える際，Σ_1 を含むようなあるアルファベット Σ_2 **に関する**補集合をとることもできる．その場合，L の補集合は $\Sigma_2^* - L$ となる．したがって，Σ_2 に関する L の補集合には，Σ_2^* に属す文字列で，Σ_2 の要素ではあるが Σ_1 の要素ではない文字を少なくとも一つ持つものはすべて含まれる．(もちろん，Σ_1 の要素だけで構成される文字列で $\Sigma_2^* - L$ に含まれるものもあり得る.) 一方，Σ_1 に関する L の補集合を考えると，$\overline{L}$ には，$\Sigma_2 - \Sigma_1$ の要素である文字を持つ文字列は一つも含まれない．したがって，正確を期すならば，補集合を考える際にはどのアルファベットに関する補集合なのかということを常に明示すべきである．しかし，通常は，どのアルファベットが対象になっているのかは断るまでもなく自明であることが多い．たとえば，L がオートマトンによって定義される言語である場合，対象となるアルファベットはオートマトンの定義の中にすでに指定されている．このような場合，我々は，アルファベットについて改めて断わることはせず，単に「補集合」といういい方をする．

図 4.2 に $\{0,1\}^* - L(A)$ を受理するオートマトンを示す．これは，一つの受理状態が非受理状態に変わり，二つの非受理状態が受理状態に変わった点を除き，図 2.14 に示したオートマトンと同じである．　▨

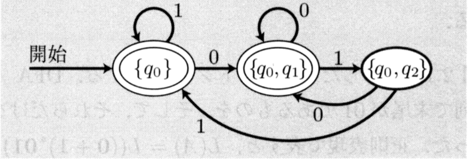


図 4.2　言語 $(\mathbf{0}+\mathbf{1})^*\ \mathbf{01}$ の補集合を受理する DFA

例 4.7　定理 4.5 を適用することによりある言語が正則ではないことを示す例を取り上げる．例 4.2 では，同じ個数の 0 と 1 を含む文字列の全体から成る言語 L_{eq} は正則言語ではないことを証明した．そこでの証明は反復補題を直接に適用するものであった．

さて，ここでは，相異なる個数の 0 と 1 を含む文字列の全体から成る言語 M を考えよう．この場合，反復補題を用いて M が正則ではないことを示すことには困難が伴う．その理由を直観的に説明してみよう．M に属するある文字列 w から出発し，それを $w = xyz$ と分解し，y を「ポンプで汲み上げる」とする．しかしこのとき，y 自身が 01 のように 0 と 1 を同じ個数ずつ持つ文字列であるかもしれない．その場合，どんな k に対しても xy^kz が同じ個数の 0 と 1 を持つことはあり得ない．そもそも xyz は相異なる個数の 0 と 1 を持ち，y を「ポンプで汲み上げる」たびに 0 と 1 の個数は同じ個数だけ変化していくからである．このような理由により，M は正則であるという仮定に対して矛盾を生じさせることを狙って反復補題を用いることには困難が伴う．

しかし，だからといって，M が正則であるという訳ではない．M が正則ではない理由は，$M = \overline{L_{eq}}$ であることによる．補集合の補集合はもとの集合に一致するので，$L_{eq} = \overline{M}$ であるが，もし M が正則であるとすると，定理 4.5 より L_{eq} も正則でなければならない．しかし，我々はすでに L_{eq} は正則ではないことを知っている．こうして，矛盾が生じたので，M は正則ではないことの背理法による証明が得られた．　▨

共通部分のもとでの閉包　次に，二つの正則言語の共通部分について考えてみよう．この場合，実は，我々はほとんど何もする必要がない．三つのブール演算は互いに独立ではないからである．言語 L と M の共通部分は，補集合をとる演算と和をとる演算を用いることが許されれば，次の等式により求めることができる．

$$L \cap M = \overline{\overline{L} \cup \overline{M}} \tag{4.1}$$

一般に，二つの集合の共通部分は，どちらの集合の補集合にも含まれない要素の全体から成る集合である．式 (4.1) はまさにそのことを述べているのだが，この性質は**ド・モルガンの法則** (De Morgan's law) と呼ばれる二つの恒等式のうちの一つである．もう一つは，和と共通部分を交換して得られる式，すなわち，

$L \cup M = \overline{\overline{L} \cap \overline{M}}$ である．

ただし，上の方法とは別に，二つの正則言語に対し，それらの共通部分を受理する DFA を直接構成することも可能である．この構成法の本質的部分は二つの DFA を並列に動かすところにあるが，この方法は他への応用も広く，それ自体，考察に値する価値がある．たとえば，我々は，二つの対象——銀行と商店——の行動の「直積」を表現するオートマトンとして図 2.3 のオートマトンを構成したが，その際この方法を用いている．**直積構成法** (product construction) と呼ばれるこの方法について，次の定理の中で厳密に述べることにしよう．

定理 4.8　L と M が正則言語であるとき，$L \cap M$ もまた正則言語である．

証明　L と M を受理するオートマトンを，それぞれ，$A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$，$A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$ とする．ここで，二つのオートマトンの入力アルファベットは共通であると仮定していることに注意されたい．L と M のアルファベットが異なる場合は Σ はそれらの集合和であるとする．直積構成法は，DFA に対してだけでなく，NFA に対しても同様に適用することができるが，話をなるべく単純にするため，ここでは A_L と A_M は DFA であると仮定する．

$L \cap M$ を受理するために，A_L と A_M の両方の動作を同時に模倣するオートマトン A を構成する．A の状態は，A_L の状態を第 1 成分とし，A_M の状態を第 2 成分とする二つの状態の対である．A の動作を説明するため，A は，いま，状態 (p, q) にあるとする．ただし，p は A_L の状態であり，q は A_M の状態である．入力文字 a に対する A_L の動きをみる．たとえば，a を読んで A_L は状態 s に移るとしよう．次に，入力 a に対する A_M の動きをみる．A_M は a を読んだ後，状態 t に移るとしよう．このとき，入力 a を読んだ後の A の状態は (s, t) であると定める．このようにして，A は A_L と A_M の両方の動きを同時に模倣していく．動作の考え方を図 4.3 に示す．

A の定義の残りの部分を定めるのは容易である．A の初期状態は，A_L の初期状態と A_M の初期状態の対 (q_L, q_M) であるとする．受理状態については，両方のオートマトンがともに受理するとき，また，そのときに限り受理するように定めたいので，A_L の受理状態 p と A_M の受理状態 q の対 (p, q) の全体を A の受理状態の集合とする．以上のことを定式化すると，A は次のように表される．

$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F_L \times F_M)$$

ただし，$\delta((p, q), a) = (\delta_L(p, a), \delta_M(q, a))$ である．

このとき，$L(A) = L(A_L) \cap L(A_M)$ が成り立つことを確かめるには，まず，$|w|$ に関する簡単な帰納法により $\hat{\delta}((q_L, q_M), w) = (\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w))$ が証明できることに注意する．A が w を受理するのは $\hat{\delta}((q_L, q_M), w)$ が受理状態の対であるときであ

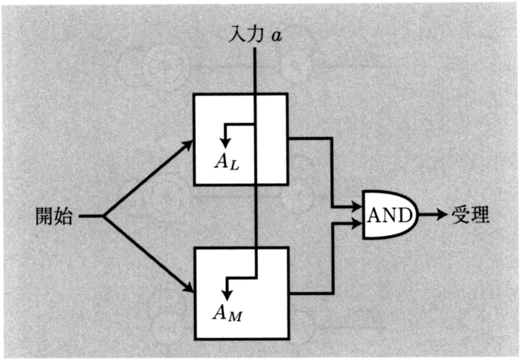


図 4.3 二つのオートマトンの動作を模倣し，両方が受理する入力を受理するオートマトン

り，また，そのときに限る．すなわち，$\hat{\delta}_L(q_L, w)$ が F_L に属し，$\hat{\delta}_M(q_M, w)$ が F_M に属すときである．いい換えると，w が A に受理されるための必要十分条件は，A_L と A_M がともに w を受理することである．よって，A は L と M の共通部分を受理する． ▨

例 4.9 図 4.4 に二つの DFA を示す．図 4.4 (a) のオートマトンは 0 を含む文字列の全体を受理する．一方，図 4.4 (b) のオートマトンは 1 を含む文字列の全体を受理する．これらの二つのオートマトンの直積を図 4.4 (c) に示す．図 4.4 (c) のオートマトンの各状態は，(a) と (b) のオートマトンの状態の対によりラベル付けされている．

ここで，(c) のオートマトンが上の二つのオートマトンで受理される言語の共通部分，すなわち，0 と 1 の両方を含む文字列の全体を受理することは容易に確かめられる．状態 pr はまだ 0 も 1 も読んでいないという初期条件を表す．状態 qr は 0 は読んだけれど 1 は読んでいないことを意味し，状態 ps は 1 は読んだけれど 0 は読んでいないことを意味する．受理状態 qs は 0 と 1 をともに読んだという条件を表す． ▨

差のもとでの閉包　集合に対して適用される演算で，ブール演算とも関係の深い第 4 の演算がある．「集合の差」の演算である．言語の言葉でいうと，L と

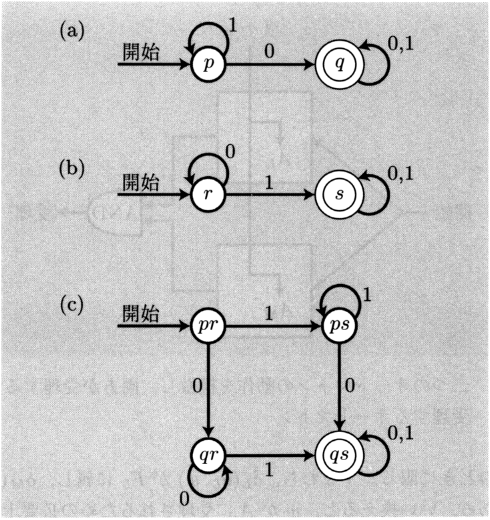


図 4.4　直積構成法

M の（集合）**差** $L-M$ とは，言語 L には属すが言語 M には属さない文字列全体の集合のことである．正則言語の族は，この演算のもとでも閉じている．その証明は，いま上で証明したばかりのいくつかの定理から容易に導くことができる．

定理 4.10　L と M が正則言語であるとき，$L-M$ もまた正則言語である．

証明　$L-M = L \cap \overline{M}$ に注意する．定理 4.5 より $\overline{M}$ は正則である．さらに，定理 4.8 より $L \cap \overline{M}$ は正則である．よって，$L-M$ は正則である．▨

4.2.2 反　転

文字列 $a_1a_2\cdots a_n$ に対し，$a_1a_2\cdots a_n$ を逆向きに並べて得られる列，すなわち，$a_na_{n-1}\cdots a_1$ を $a_1a_2\cdots a_n$ の**反転** (reversal) という．文字列 w の反転を w^R と表す．たとえば，0010^R は 0100 であり，ε^R は ε である．

言語 L の反転とは，L に属す文字列の反転の全体から成る言語である．これを L^R と表す．たとえば，$L=\{001, 10, 111\}$ とすると $L^R=\{100, 01, 111\}$ で

ある．

反転の演算もまた，言語の正則性を保存する．すなわち，L が正則言語であるとき，L^R もまた正則言語である．この性質に対しては，二つの簡単な証明がある．一つはオートマトンを用いるものであり，もう一つは正則表現を用いるものである．ここではまず，オートマトンに基づく証明についてその大筋を示す．興味ある読者は，説明し残された細部を埋めて証明を完成させてみていただきたい．次に，正則表現を用いた証明を，これは細部まできちんと述べる．

言語 L は，ある有限オートマトン A に対して $L = L(A)$ となる言語であるとする．A は非決定性であってもよいし，ε-動作つきであってもよい．A を次のように変形することにより L^R を受理するオートマトンを構成することができる．

(1) A に対する状態遷移図において有向辺の矢印の向きをすべて逆向きに付けかえる．

(2) A の初期状態を新しいオートマトンのただ一つの受理状態とする．

(3) 新しい初期状態 p_0 を作り，p_0 から A のすべての受理状態へ ε-動作の遷移を導入する．

このようにして得られるオートマトンは，A を「逆向き」に模倣するオートマトンであり，したがって，このオートマトンが文字列 w を受理するのは A が w^R を受理するときであり，またそのときに限る．

さて，次に「反転の定理」に対する厳密な証明を与えよう．

定理 4.11 L が正則言語であるとき，L^R も正則言語である．

証明 L は正則表現 E によって定義される言語であるとする．証明は，E の長さに関する構造的帰納法による．我々は，$L(E^R) = (L(E))^R$ を満たす正則表現 E^R が存在することを示す．このとき，正則表現 E^R に対応する言語は正則表現 E に対応する言語の反転である．

基礎：E が ε，$\emptyset$，または，ある文字 a に対し $\mathbf{a}$ であるとき，$\{\varepsilon\}^R = \{\varepsilon\}$，$\emptyset^R = \emptyset$，$\{a\}^R = \{a\}$ であるので，E^R は E と同じでよい．

帰納：E の形により，三つの場合に分けられる．

(1) $E = E_1 + E_2$ の場合．このとき，$E^R = E_1^R + E_2^R$ とすればよい．その理由は，二つの言語の和の反転は，各々の言語を反転し，それらの和を求めることにより得られるからである．

(2)　$E = E_1E_2$ の場合．このとき，$E^R = E_2^R E_1^R$ とすればよい．ここで，二つの表現の各々を反転させるだけでなく，それらの順序も逆転させていることに注意されたい．たとえば，$L(E_1) = \{01, 111\}$，$L(E_2) = \{00, 10\}$ のとき，$L(E_1E_2) = \{0100, 0110, 11100, 11110\}$ であり，この言語の反転は

$$\{0010, 0110, 00111, 01111\}$$

である．一方，$L(E_2)$ と $L(E_1)$ の各々の反転をこの順に連接すると

$$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$

が得られる．これは $(L(E_1E_2))^R$ と同じ言語である．一般に，$L(E)$ に属す語 w が $L(E_1)$ に属す w_1 と $L(E_2)$ に属す w_2 の連接であるとき，$w^R = w_2^R w_1^R$ が成り立つ．

(3)　$E = E_1^*$ の場合．このとき，$E^R = (E_1^R)^*$ とすればよい．その理由は次の通りである．$L(E)$ に属す任意の文字列 w は $w_1w_2 \cdots w_n$ と表される．ただし，各々の w_i は $L(E_1)$ に属すとする．このとき，

$$w^R = w_n^R w_{n-1}^R \cdots w_1^R$$

となる．各々の w_i^R は $L(E^R)$ に属し，したがって w^R は $(E_1^R)^*$ に属す．逆に，$L((E_1^R)^*)$ に属す任意の文字列は $w_1w_2 \cdots w_n$ の形で表される．ただし，各々の w_i は $L(E_1)$ に属す文字列の反転である．この文字列の反転，$w_n^R w_{n-1}^R \cdots w_1^R$，は，したがって $L(E_1^*)$ に属す文字列であり，すなわち $L(E)$ に属す．以上より，任意の文字列が $L(E)$ に属すための必要十分条件は，その反転が $L((E_1^R)^*)$ に属すことであることが示された．▨

例 4.12　正則表現 $(\mathbf{0}+\mathbf{1})\mathbf{0}^*$ によって定義される言語を L とする．このとき，L^R は，連接に関する規則により，$(\mathbf{0}^*)^R(\mathbf{0}+\mathbf{1})^R$ が表す言語である．ここで二つの部分にそれぞれ閉包に関する規則と和に関する規則を適用し，さらに，**0** と **1** の反転は各々それ自身である，という帰納法の基礎に記した規則を適用すると，L^R は正則表現 $\mathbf{0}^*(\mathbf{0}+\mathbf{1})$ に対応する言語であることがわかる．

4.2.3 準同型写像

文字列に対する**準同型写像** (homomorphism) とは，文字列上で定義される関数（写像）であり，文字列の中の各文字を特定の文字列で置き換える（各文字に特定の文字列を代入する）ものである．

例 4.13　アルファベット $\{0, 1\}$ 上で $h(0) = ab$ と $h(1) = \varepsilon$ とによって定義される関数 h は準同型写像である．0 と 1 とから成る任意の文字列が与えられ

たとき，h はすべての 0 を文字列 ab によって置き換え，すべての 1 を空列によって置き換える．たとえば，h を文字列 0011 に適用すると $abab$ という列が得られる．▨

一般に，h がアルファベット Σ 上の準同型写像であり，$w = a_1 a_2 \cdots a_n$ が Σ に属す文字からなる列であるとき，$h(w) = h(a_1)h(a_2)\cdots h(a_n)$ が成り立つ．すなわち，w の中の各々の文字に h を施し，その結果を同じ順序で連接して得られる列が $h(w)$ である．たとえば，h を例 4.13 に示した準同型写像とするとき，$w = 0011$ に対し $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\varepsilon)(\varepsilon) = abab$ となる．これは例 4.13 に記した通りである．

さらに，言語に対しても準同型写像を施すことができる．言語に属す文字列のすべてに準同型写像を施すのである．すなわち，L をアルファベット Σ 上の言語とし，h を Σ 上の準同型写像とするとき，$h(L) = \{h(w) \mid w \in L\}$ と定める．たとえば，h を例 4.13 で定めた準同型写像とし，L を正則表現 **10*1** に対応する言語，すなわち，二つの 1 の間に任意の個数の 0 がはさまれているような列の全体であるとするとき，$h(L)$ は $(\mathbf{ab})^*$ に対応する言語である．なぜならば，例 4.13 の h は 1 を ε で置き換えるので，つまり 1 を無視してしまえばよく，またすべての 0 を ab で置き換えればよいからである．これと同様のアイデアを用いて，準同型写像を正則表現に直接施すことにより，正則言語の族は準同型写像のもとで閉じていることを証明することができる．

定理 4.14　L がアルファベット Σ 上の正則言語であり，h が Σ 上の準同型写像であるとき，$h(L)$ も正則言語である．

証明　ある正則表現 R に対し $L = L(R)$ とする．一般に，E を Σ に属す文字から成る正則表現とするとき，E において Σ に属す各文字 a を $h(a)$ によって置き換えて得られる正則表現を $h(E)$ とする．以下，我々は，$h(L)$ は $h(R)$ に対応する言語であることを示す．

証明は，簡単な構造的帰納法を用いて行われる．次のことを示す．R の任意の部分表現 E をとり，それに h を施して得られる表現を $h(E)$ とするとき，$h(E)$ に対応する言語は，言語 $L(E)$ に h を施して得られる言語と同じである．

基礎：E が ε または $\varnothing$ のとき，h は ε や $\varnothing$ には影響を与えないので，$h(E)$ は E と同じである．したがって，$L(h(E)) = L(E)$ である．一方，E が $\varnothing$ または ε のとき，$L(E)$ は，それぞれ，文字列を一つも含まない集合であるか，または，文字を一つも含

まない列を一つ含むだけの集合である．したがって，いずれの場合も $h(L(E)) = L(E)$ となる．すなわち，$L(h(E)) = L(E) = h(L(E))$ が結論される．

基礎の段階でもう一つ残されているのは，Σ に属するある文字 a に対し $E = \mathbf{a}$ となる場合である．この場合は，$L(E) = \{a\}$ であり，したがって，$h(L(E)) = \{h(a)\}$ である．また，$h(E)$ は文字列 $h(a)$ として表される正則表現である．したがって，$L(h(E))$ もまた $\{h(a)\}$ であり，$L(h(E)) = h(L(E))$ が結論される．

帰納：三つの場合に分けられるが，いずれの場合も簡単である．ここでは，和の場合，すなわち，$E = F + G$ となる場合だけを証明することにする．準同型写像の正則表現への適用のしかたから $h(E) = h(F + G) = h(F) + h(G)$ であることが保証される．また，$L(E) = L(F) \cup L(G)$ であり，

$$L(h(E)) = L(h(F) + h(G)) = L(h(F)) \cup L(h(G)) \tag{4.2}$$

である．これは，正則表現における "+" の意味から明らかである．最後に，

$$h(L(E)) = h(L(F) \cup L(G)) = h(L(F)) \cup h(L(G)) \tag{4.3}$$

が導かれる．これは，h を言語に施すことは，その言語に属す個々の文字列に独立に h を施すことと同じであることから明らかである．さて，ここで帰納法の仮定を用いると $L(h(F)) = h(L(F))$ および $L(h(G)) = h(L(G))$ が得られる．したがって，式 (4.2) の最後の項と式 (4.3) の最後の項は同じである．よって，これらの 2 式の先頭の二つの項も相等しい．すなわち，$L(h(E)) = h(L(E))$ である．

ここでは，表現が連接である場合や閉包である場合の証明は記さない．ただし，どちらの場合も証明のアイデアは上と同様である．

このようにして，$L(h(R))$ は実際 $h(L(R))$ に等しいことが結論される．すなわち，言語 L を定義する正則表現に準同型写像 h を施すとき，得られる結果は言語 $h(L)$ を定義する正則表現である．　▨

4.2.4 逆準同型写像

準同型写像は「逆向きに（すなわち，逆写像として）」適用することも可能である．そして，その場合も正則言語の族は保存される．まず，h をあるアルファベット Σ から別の（同じでもよい）アルファベット T [2]上の文字列への準同型写像とする．L をアルファベット T 上の言語とする．このとき，$h^{-1}(L)$ は，Σ^* に属す文字列 w で，$h(w)$ が L に属すようなものの全体から成る集合である．なお，$h^{-1}(L)$ は「L の h による逆像（L の h インバース）」と読む．図 4.5 の (a) に言語 L 上の準同型写像の様子を，また，(b) に逆準同型写像の

2) ここで使われている T は，ギリシャ文字で「シグマ」の次に位置する文字「タウ」の大文字であるとご理解いただきたい．

様子を図示する．

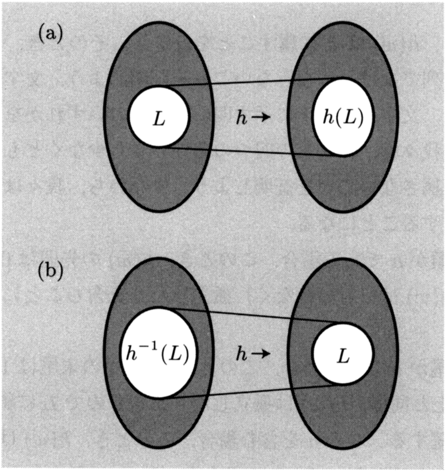


図 4.5　準同型写像と逆準同型写像

例 4.15　L を正則表現 $(\mathbf{00}+\mathbf{1})^*$ に対応する言語とする．すなわち，L は，0 と 1 とから成る文字列で，0 はすべて対として（二つずつ）出現するものの全体から成る言語である．したがって，0010011 や 10000111 は L に属し，000 や 10100 は L に属さない．

さて，h を $h(a)=01$ および $h(b)=10$ によって定義される準同型写像であるとする．このとき，$h^{-1}(L)$ は正則表現 $(\mathbf{ba})^*$ に対応する言語であること，すなわち ba という対が何回か繰り返し現れる文字列の全体であることを示す．以下では，$h(w)$ が L に属すための必要十分条件は w が $baba\cdots ba$ の形をしていることである，ということを証明する．

（十分性）　ある $n\geq 0$ に対し，w は ba が n 回繰り返し出現している列 $baba\cdots ba$ であるとする．ここで，$h(ba)=1001$ であるので，$h(w)$ は 1001 を n 回繰り返して得られる列であることに注意する．文字列 1001 は，二つの 1 と 1 組の 0 の対とで構成されている．したがって，明らかに，1001 は L に属す．さらに，1001 を何回か繰り返して得られる任意の文字列も 1 と 00 とを部品として

構成されているので，同様に L に属す．したがって，$h(w)$ は L に属すことがわかる．

（必要性） 次に，$h(w)$ は L に属すことを仮定し，そのとき，w は $baba\cdots ba$ の形をした文字列でなければならないことを証明しよう．文字列が上のような形ではない場合，文字列は以下に示す四つの条件のいずれかを満たしていなければならない．我々は，これらの四つの条件のうち少なくとも一つが成り立てば $h(w)$ は L に属さないことを証明しよう．すなわち，我々は，証明したい命題の対偶を証明することになる．

(1) w の先頭が a である場合．このとき，$h(w)$ の先頭は 01 である．したがって，$h(w)$ は（対ではなく）孤立した 0 を含むことになり，L に属さない．

(2) w の末尾が b である場合．このとき，$h(w)$ の末尾は 10 である．したがって，上と同様，$h(w)$ は孤立した 0 を含むので L に属さない．

(3) w が連続する二つの a を含む場合．このとき，$h(w)$ は 0101 を部分列として含む．したがって，この場合も，$h(w)$ は孤立した 0 を含むので L に属さない．

(4) w が連続する二つの b を含む場合．このとき，$h(w)$ は 1010 を部分列として含む．したがって，上と同様，$h(w)$ は孤立した 0 を含むので L に属さない．

したがって，上の四つの条件のうち少なくとも一つが成り立つとき，$h(w)$ は L に属さない．一方，上の四つの条件がどれも成り立たないならば，w は $baba\cdots ba$ の形をした文字列である．そのことを見るために，条件 (1)〜(4) がどれも成り立たないと仮定してみよう．すると，(1) より，w の先頭は b でなければならないことになり，また，(2) より，w の末尾は a でなければならないことになる．さらに，(3) と (4) より，w の中で a と b は交互に出現しなければならない．したがって，四つの命題 (1) $\sim$ (4) の論理和 (OR) は，「w は $baba\cdots ba$ の形をしていない」という命題と同値である．我々はすでに，(1) $\sim$ (4) の論理和 (OR) から「$h(w)$ は L に属さない」ことが導かれることを示した．これらの命題より，証明すべき命題「$h(w)$ が L に属すならば w は $baba\cdots ba$ の形をしている」の対偶が得られたことになる． ▨

次に，正則言語の準同型写像の逆像は，やはり，正則言語であることを証明しよう．そのあとで，この定理の応用について述べることにする．

定理 4.16 アルファベット Σ からアルファベット T への準同型写像を h とする．L が T 上の正則言語であるならば，$h^{-1}(L)$ も Σ 上の正則言語である．

証明　オートマトンを用いて証明を行うため，まず，L を受理する DFA A を定める．図 4.6 に示すような方針で，A と h とから $h^{-1}(L)$ を受理する DFA を構成する．目的とする DFA は，A と同じ状態集合を持つが，次に遷移する状態を決定する前に h を用いて入力文字を変換する．

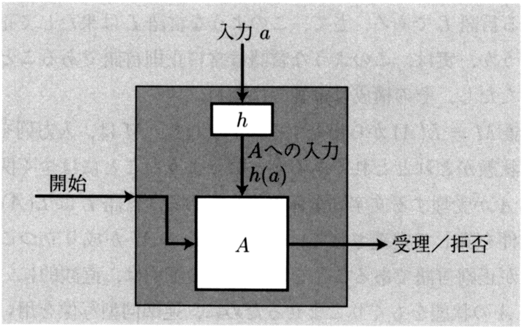


図 4.6　$h^{-1}(L)$ を受理する DFA は，まず，入力に h を施し，次に，L を受理する DFA の動きを模倣する．

形式的に書くと，次のようになる．DFA $A = (Q, T, \delta, q_0, F)$ に対し $L = L(A)$ とする．DFA B を

$$B = (Q, \Sigma, \gamma, q_0, F)$$

と定める．ただし，遷移関数 γ は $\gamma(q, a) = \hat{\delta}(q, h(a))$ という規則にしたがって定義される．すなわち，B が入力 a に対して行う遷移は，A が文字列 $h(a)$ の各文字に対して順に行う一連の遷移の結果に等しい．なお，$h(a)$ は ε の場合もあり得るし，1 文字の場合もあり得るし，また，数個の文字から成る文字列の場合もあり得ることに注意されたい．$\hat{\delta}$ は，これらのすべての場合に対して正しく対処できるように，適切に定義されている．

このとき，$\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$ が成り立つことは $|w|$ に関する簡単な帰納法により示すことができる．A と B の受理状態の集合は同一なので，B が w を受理するための必要十分条件は A が $h(w)$ を受理することである．別のいい方をすると，B は

$h^{-1}(L)$ に属するすべての文字列を，そしてそれらだけを，受理する． ▨

例 4.17 この例では，逆準同型写像と，正則集合が持ついくつかの閉包性とを利用して，有限オートマトンに関する一つの奇妙な事実を証明しよう．DFA が入力を受理するための条件として，すべての状態を少なくとも 1 回は訪れないといけない，という付加条件を課すことを要求したとする．もう少し厳密にいうと，我々が興味を持つのは，$A = (Q, \Sigma, \delta, q_0, F)$ を DFA とするとき，Σ^* に属す文字列 w で，$\hat{\delta}(q_0, w)$ は F に属し，かつ，Q に属す任意の q に対して $\hat{\delta}(q_0, x_q) = q$ を満たす w の接頭語 x_q が存在するという性質を持つすべての文字列からなる言語 L である．さて，このような言語 L は果たして正則言語となるのであろうか．実は，このような言語は常に正則言語であることを示すことができる．ただし，その構成は単純ではない．

まず，言語 $M = L(A)$ から出発する．すなわち，M は，入力列を処理する過程で訪れる状態がどれとどれであるかというようなことには全く関知せず，通常の意味で A が受理する文字列全体の集合である．言語 L は $L(A)$ に属す文字列に付加条件を課して定義されているので，$L \subseteq M$ が成り立つことに注意されたい．L が正則言語であることを示す我々の証明は，直観的にいうと，入力文字の中に A の状態をもぐりこませるために，逆準同型写像を用いることから始まる．正確にいうと，まず，新しいアルファベット T を定める．T は，以下のような意味を持つ三つの要素の組 $[paq]$ とみなすことのできる文字から成るアルファベットである．

(1) p と q は Q に属す状態である．

(2) a は Σ に属す文字である．

(3) $\delta(p, a) = q$.

すなわち，T に属す各文字は，オートマトン A の遷移を表していると考えることができる．ただし，ここで大事なことは，$[paq]$ という表記は，三つの文字の連接を表すのではなく，一つの文字の（我々流の）表現だということである．もちろん，このような表記法を採らずに単独の文字を用いることにしてもよい．しかしそのようにすると，その文字と p, q, a との関係を記述することは少なからず困難になる．

さて，準同型写像 h を，任意の p, q, a に対し $h([paq]) = a$ として定めよう．

すなわち，h は，T に属す文字の各々から Σ に属す文字のみを残して，状態を表す部分を取り除く働きをする．L が正則言語であることを示す我々の最初のステップは，言語 $L_1 = h^{-1}(M)$ を構成することである．M は正則なので，定理 4.16 より，L_1 も正則である．L_1 に属す文字列とは，M に属す文字列に，その中の各文字ごとに，遷移を表す状態の対を付加して得られるものに他ならない．

理解を助けるため，非常に簡単な例として図 4.4(a) に示した 2 状態のオートマトンを考えてみたい．アルファベット Σ は $\{0,1\}$ であり，アルファベット T は四つの文字 $[p0q]$, $[q0q]$, $[p1p]$, $[q1q]$ からなる．たとえば，入力 0 に対し，状態 p から状態 q への遷移があるので，$[p0q]$ は T の一つの文字である．文字列 101 はこのオートマトンによって受理される．文字列 101 に h^{-1} を施すと $2^3 = 8$ 個の文字列が得られる．たとえば，$[p1p][p0q][q1q]$ や $[q1q][q0q][p1p]$ がその例である．

さて次に，正則言語を保存する一連の演算を用いて，L_1 から L を構成しよう．最初の目標は，L_1 の文字列の中で状態に関する扱いが正しくないものを取り除くことである．前に述べたように，$[paq]$ という文字を，オートマトンは状態 p にあり，入力 a を読み，状態 q に入るということを表すものとみなす．一連の文字の列は，それが A の受理に至る計算を導くものであるとみなすことができるためには，以下の三つの条件を満たす必要がある．

(1)　最初の文字の第 1 の状態は，A の初期状態 q_0 である．

(2)　各遷移は一つ前の遷移を引き継ぐものとなっている．すなわち，(2 番目以降の）任意の文字の第 1 の状態は，一つ前の文字の第 2 の状態に一致している．

(3)　最後の文字の第 2 の状態は，F に属す状態である．

L を構成する手順の概略を図 4.7 に示す．

条件 (1) は，任意の文字 a と任意の状態 q に対し $[q_0aq]$ の形の文字で始まる文字列全体の集合と L_1 との共通部分をとることで実現される．すなわち，E_1 として正則表現 $[q_0a_1q_1] + [q_0a_2q_2] + \cdots$ をとる．ただし，a_iq_i は $\Sigma \times Q$ に属し $\delta(q_0, a_i) = q_i$ を満たす対 (a_i, q_i) の全体にわたる．この E_1 に対し，$L_2 = L_1 \cap L(E_1T^*)$ とする．E_1T^* は，T^* に属す文字列で初期状態で始まるものの全体を表す正則表現（ただし，正則表現の中で T と記すとき，T は T に属す文字全体の和であるとみなすことにする）なので，L_2 は，言語 M に h^{-1}

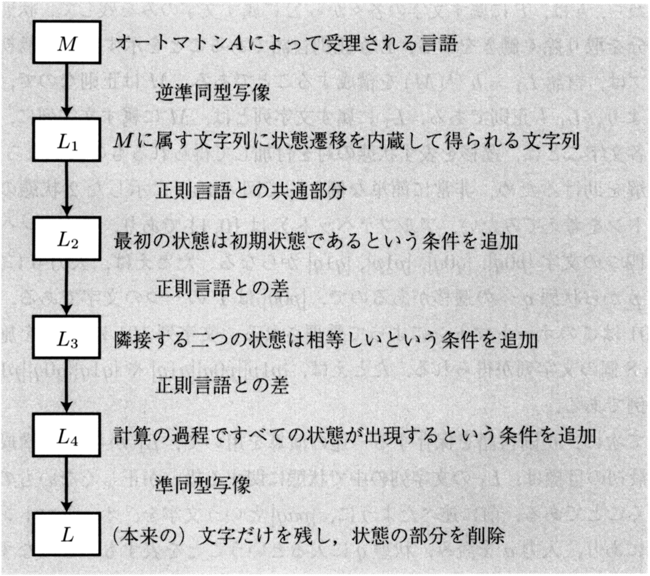


図 4.7 言語の正則性を保存する演算を施すことにより，言語 M から言語 L を構成する方法

を施して得られる形で，先頭の文字の第 1 成分として初期状態を持つような文字列全体の集合となる．すなわち，L_2 は条件 (1) を満たす．

条件 (2) を実現するには，L_2 から（集合差の演算を用いて）条件 (2) に反するすべての文字列を引き去るのが容易な方法である．E_2 を，正しく適合しない文字の対，すなわち，$q \neq r$ に対し $[paq][rbs]$ の形の連接全体の和（集合和）からなる正則表現とする．このとき，$T^*E_2T^*$ は，条件 (2) に合致しない文字列の全体を表す正則表現となる．

次に，$L_3 = L_2 - L(T^*E_2T^*)$ と定める．そうすると，L_3 に属す文字列は，条件 (1) を満たす．L_2 に属す文字列はどれも先頭が初期状態を含む文字となっているからである．さらに，L_3 に属す文字列は条件 (2) も満たす．$L(T^*E_2T^*)$ との差をとることで，条件 (2) に反する文字列はすべて取り除かれるからであ

る．最後に，L_3 に属す文字列は，条件 (3) を満たす．すなわち，最後の文字の2番目の状態は受理状態である．なぜならば，我々はそもそも M に属す文字列，すなわち，A を受理状態に導く文字列だけを対象として考察を始めたからである．以上をまとめると，L_3 は，M に属す文字列で，その受理に至る計算過程に出現するすべての状態が各文字の一部として埋め込まれたようなものの全体からなるものであることがわかる．なお，L_3 は正則言語であることに注意されたい．L_3 は，正則言語 M から出発し，正則言語に対して適用されると再び正則言語を生じるようないくつかの演算——逆準同型写像，共通部分，集合差——を施した結果として得られる言語であるからである．

さて，我々の目標は，M に属す文字列で，受理に至る計算の過程ですべての状態を訪れるような文字列のみを受理することであった．我々は，この条件を満たすようにするため，集合差の演算をもう一度適用することにする．任意の状態 q に対し，E_q を，T に属すすべての文字で，第1の状態にも第2の状態にも q が現れないようなものの全体の和を表す正則表現とする．L_3 から $L(E_q^*)$ を引くと，A の受理に至る計算で，状態 q を少なくとも1回訪れるものに対応するすべての文字列の集合が残る．L_3 から，Q の任意の要素 q に対する言語 $L(E_q^*)$ をすべて引いてしまうと，A の受理に至る計算で，どの状態も少なくとも1回は訪れるものに対応する文字列全体から成る言語が得られる．この言語を L_4 とおく．定理 4.10 により，L_4 もやはり正則である．

最後に残された作業は，L_4 から，状態を表す部分を完全に取り去って L を構成することである．このためには，$L = h(L_4)$ とすればよい．こうすれば，L は，Σ^* に属する文字列で，A によって受理され，かつ，受理に至る計算の過程で A のどの状態も少なくとも1回は訪れるような文字列全体の集合となる．正則言語の族は準同型写像のもとで閉じていることから，L は正則言語であることが結論される． ▨

4.2.5 4.2 節の演習問題

♣ **問 4.2.1** アルファベット $\{0, 1, 2\}$ からアルファベット $\{a, b\}$ への準同型写像 h を次のように定める: $h(0) = a$, $h(1) = ab$, $h(2) = ba$.

*a) $h(0120)$ を求めよ．

b) $h(21120)$ を求めよ．

*c) L を言語 $L(\mathbf{01^*2})$ とするとき，$h(L)$ を求めよ．

d) L を言語 $L(\mathbf{0}+\mathbf{12})$ とするとき，$h(L)$ を求めよ．

*e) L を言語 $\{ababa\}$（一つの文字列 $ababa$ だけから成る言語）とするとき，$h^{-1}(L)$ を求めよ．

!f) L を言語 $L(\mathbf{a}(\mathbf{ba})^*)$ とするとき，$h^{-1}(L)$ を求めよ．

♣*! **問 4.2.2** 言語 L と文字 a に対し，wa が L に属すような文字列 w の全体から成る集合を a に関する L の**商** (quotient) といい，L/a と表す．たとえば，$L = \{a, aab, baa\}$ のとき $L/a = \{\varepsilon, ba\}$ である．L が正則ならば L/a も正則であることを証明せよ．**ヒント**：L を受理する DFA から出発し，受理状態の集合を考察せよ．

♣ ! **問 4.2.3** 言語 L と文字 a に対し，aw が L に属すような文字列 w の全体から成る集合を $a\backslash L$ と表す．たとえば，$L = \{a, aab, baa\}$ のとき $a\backslash L = \{\varepsilon, ab\}$ である．L が正則ならば $a\backslash L$ も正則であることを証明せよ．**ヒント**：正則言語の族は反転をとる演算のもとで閉じているし，また，問 4.2.2 より，商をとる演算のもとでも閉じている．

♣ ! **問 4.2.4** 次の等式のうち，正しいのはどれであるか答えよ．

a) $(L/a)a = L$ （左辺は二つの言語 L/a と $\{a\}$ の連接を表す）

b) $a(a\backslash L) = L$ （ここでも，左辺は $\{a\}$ と a/L の連接を表す）

c) $(La)/a = L$

d) $a\backslash(aL) = L$

♣ **問 4.2.5** 問 4.2.3 で定義された演算は「微分」と見なされ，$a\backslash L$ は $\dfrac{dL}{da}$ と表されることがある．この微分は正則表現に対しても適用される．適用のしかたは，通常の微分を算術式に対して適用する場合と同様である．R を正則表現とし，$L = L(R)$ とするとき，$\dfrac{dR}{da}$ は $\dfrac{dL}{da}$ と同じものを表すとする．

a) $\dfrac{d(R+S)}{da} = \dfrac{dR}{da} + \dfrac{dS}{da}$ を示せ．

*!b) 連接 RS に対する「微分」の公式を求めよ．**ヒント**：$L(R)$ が ε を含む場合と含まない場合の二つの場合に分けて考える必要がある．ここで求める公式は，通常の微分に対する「積の公式」と，似てはいるが，まったく同じ形ではない．

!c) 閉包の「微分」，すなわち，$\dfrac{d(R^*)}{da}$ に関する公式を求めよ．

d) 上記 (a)～(c) の公式を用いて，正則表現 $(\mathbf{0}+\mathbf{1})^*\mathbf{011}$ の 0 に関する「微分」と 1 に関する「微分」を求めよ．

*e) $\dfrac{dL}{d0} = \varnothing$ を満たす言語 L の特徴づけを行え．(どのような言語であるかを示せ．)

*!f) $\dfrac{dL}{d0} = L$ を満たす言語 L の特徴づけを行え．

♣ !**問 4.2.6**　正則言語の族は，次の各々の演算のもと閉じていることを示せ．

a) $min(L) = \{w \mid w$ は L に属すが，w の真の接頭語は L に属さない$\}$

b) $max(L) = \{w \mid w$ は L に属すが，ε 以外のすべての x に対し wx は L に属さない$\}$

c) $init(L) = \{w \mid$ ある x に対し, wx は L に属す$\}$

ヒント：問 4.2.2 と同様に，L を受理する DFA から出発し，目的の言語を得るための適切な構成を試みるのがもっとも容易な方法であろう．

♣ !**問 4.2.7**　同じ長さの二つの文字列 $w = a_1a_2\cdots a_n$ と $x = b_1b_2\cdots b_n$ に対し，w と x に含まれる文字を，w の方から先に，交互に並べて得られる文字列 $a_1b_1a_2b_2\cdots a_nb_n$ を $alt(w, x)$ と定める．言語 L と M に対し，L に属す任意の文字列 w と，M に属し w と同じ長さを持つ任意の文字列 x に対する $alt(w, x)$ の形の文字列全体の集合を $alt(L, M)$ と定める．L と M がともに正則ならば，$alt(L, M)$ も正則であることを証明せよ．

♣!!! **問 4.2.8**　L を任意の言語とする．L に属す文字列の前半分をすべて集めた集合，すなわち，$\{w \mid |x| = |w|$ を満たすある x に対し, wx は L に属す$\}$ を $half(L)$ と定める．たとえば，$L = \{\varepsilon, 0010, 011, 010110\}$ とすれば，$half(L) = \{\varepsilon, 00, 010\}$ である．なお，長さが奇数の文字列は $half(L)$ の決定には一切関与しないことに注意されたい．L が正則言語であるならば，$half(L)$ も正則言語であることを証明せよ．

♣!! **問 4.2.9**　問 4.2.8 を，文字列の中から取り出す割合を決める関数をいろいろに変えることにより，一般化することが可能である．整数に対して定義される関数 f に対し，言語 $f(L)$ を $\{w \mid |x| = f(|w|)$ を満たすある x に対し, wx は L に属す$\}$ として定める．たとえば，f を恒等関数 $f(n) = n$ とすると，f は演算 $half$ に対応する．$half(L)$ は $|x| = |w|$ によって定義されるからである．以下の各関数 f に対し，L が正則言語であるならば，$f(L)$ も正則言語であることを証明せよ．

a) $f(n) = 2n$（先頭の 3 分の 1 を取り出す）

b) $f(n) = n^2$（取り出す部分の長さは，取り残す部分の長さの平方根に等しい）

c) $f(n) = 2^n$（取り出す部分の長さは，取り残す部分の長さの対数に等しい）

♣!! **問 4.2.10**　L をアルファベット $\{0\}$ 上の任意の言語とする．すなわち，L に属す文字列は 0 だけから成る列である．ただし，L は必ずしも正則言語とは限らない．このとき，L^* は常に正則言語であることを示せ．**ヒント**：一見すると，この定理は奇妙なものに思える．具体的な例について考えてみることが，定理が正し

い理由を理解する助けになるであろう．言語 $L = \{0^i \mid i \text{ は素数 }\}$ について考えてみよう．例 4.3 により，この言語は正則ではない．さて，2 と 3 はいずれも素数であるので，文字列 00 と 000 は L に属す．したがって，$j \geq 2$ に対し 0^j は L^* に属す．j が偶数のときは 00 のコピーを $j/2$ 個使い，j が奇数のときは 000 のコピーを 1 個と 00 のコピーを $(j-3)/2$ 個使えばよい．このような考察により，$L^* = \varepsilon + \mathbf{000}^*$ であることがわかる．

♣ !! **問 4.2.11** 正則言語の族は，次に記す演算 *cycle* のもとで閉じていることを示せ：

$$cycle(L) = \{w \mid yx \text{ が } L \text{ に属すようなある } x, y \text{ に対し } w = xy\}$$

たとえば，$L = \{01, 011\}$ とすると，$cycle(L) = \{01, 10, 011, 110, 101\}$ である．**ヒント**：L を受理する DFA から出発し，$cycle(L)$ を受理する ε-NFA を構成せよ．

♣ !! **問 4.2.12** $w_1 = a_0a_0a_1$ とし，任意の $i > 1$ に対し $w_i = w_{i-1}w_{i-1}a_i$ とする．たとえば，$w_3 = a_0a_0a_1a_0a_0a_1a_2a_0a_0a_1a_0a_0a_1a_2a_3$ である．$L_n = \{w_n\}$ とする．すなわち，L_n はただ一つの文字列 w_n から成る言語である．L_n を表す正則表現で長さが最小のものは文字列 w_n 自身であり，その長さは $2^{n+1} - 1$ である．ところが，演算子として共通部分をとる演算子の使用も認めることにすると，L_n を表す正則表現で，長さが $O(n^2)$ であるものが存在する．そのような正則表現を求めよ．**ヒント**：それぞれが長さ $O(n)$ の正則表現に対応する n 個の言語で，それら全体の共通部分が L_n であるものを見つけよ．

♣ ! **問 4.2.13** 閉包性の助けを借りて，ある特定の言語が正則でないことを証明することができる．言語

$$L_{0n1n} = \{0^n1^n \mid n \geq 0\}$$

は正則集合ではないという事実から出発する．次のそれぞれの言語について，正則性を保存することが知られている演算を施してそれらを L_{0n1n} に変換することにより，その言語が正則ではないことを証明せよ．

- *****a) $\{0^i1^j \mid i \neq j\}$
- b) $\{0^n1^m2^{n-m} \mid n \geq m \geq 0\}$

♣ **問 4.2.14** 定理 4.8 では，二つの DFA に対し，それらが受理する言語の共通部分を受理する DFA を構成する方法として「直積構成法」を取り上げ，説明した．

- a) （ε-動作を含まない）NFA に対して「直積構成法」を適用する方法を示せ．
- **!**b) ε-NFA に対して「直積構成法」を適用する方法を示せ．
- *****c) 与えられた二つの DFA に対し，それらが受理する言語の差を受理する DFA を得るようにするには「直積構成法」をどのように修正すればよいかを記せ．
- d) 与えられた二つの DFA に対し，それらが受理する言語の和を受理する DFA を得るようにするには「直積構成法」をどのように修正すればよいかを記せ．

♣ **問 4.2.15**　定理 4.8 の証明において，w の長さに関する帰納法によって

$$\hat{\delta}((q_L, q_M), w) = (\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w))$$

を証明することができると述べた．帰納法による証明を記せ．

♣ **問 4.2.16**　定理 4.14 の証明について，正則表現が二つの正則表現の連接である場合と，ある正則表現の閉包である場合とに対する証明をつけ加えることにより，定理の証明を完成せよ．

♣ **問 4.2.17**　定理 4.16 の証明において，$\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$ という式の，w の長さに関する帰納法による証明を省略した．この式を証明せよ．

4.3　正則言語に関する決定問題

本節では，正則言語に関するある種の重要な質問にどう答えるかという問題を考えていく．しかし，まずその前に，言語について質問をするということはどういうことなのか，その意味を考える必要がある．代表的な言語は，大抵，無限集合である．言語が無限集合である場合，質問相手に対し，言語に属するすべての文字列を提示して，それらの文字列の点検を迫るような質問をすることはできない．通常は，そのかわりに，言語の有限的な表現を示すことによって言語の提示を行う．有限的な表現とは，たとえば，我々がすでに学んできたDFA，NFA，ε-NFA，そして，正則表現などのことである．

もちろん，上に例示した表現を使う場合，得られる言語は正則言語に限られる．実際，完全に任意に与えられる言語に対して適用可能であるような有限的な表現方法は存在しない．後の章では，正則言語の族よりも大きな言語の族に対し，言語の有限的な表現方法を与える．それらの表現を使えば，正則言語の族よりもっと大きな族に属する言語についても，その言語に関する質問を発することができるようになる．ただし，あらかじめ断わっておくが，我々が問う質問の多くについては，正則言語の族に対してのみアルゴリズムが存在する．同じ質問が，正則言語のために用意された表現よりももっと「強力な」表現（もっと大きな言語のクラスに対して適用可能な表現）に対して問われると「決定不能 (undecidable)」（その質問に答えるアルゴリズムは存在しないこと）になってしまうことが多い．

以下，正則言語に関するある種の質問に対しそれに答えるアルゴリズムを考えていく．その学習の手始めとして，言語の一つの表現を別の表現に変換する

方法の復習から始める．その際，変換を実行するアルゴリズムの時間計算量について，特に注意を払って考察していきたい．その後に，言語に関する次のような基本的な問題を考えていく．

a) 与えられた言語は空言語であるか?

b) ある文字列 w は与えられた言語に属すか?

c) 言語の二つの表現が与えられたとき，それらは同じ言語を表すか? この問題は，言語の「等価性問題」と呼ばれることがある．

4.3.1 異なる表現の間の変換

正則言語に対する四つの表現のどれについても，それを他の三つの表現の任意のものに変換することが可能であることを，我々はすでに知っている．図 3.1 に，任意の表現から他の任意の表現への変換の道筋を示した．四つの表現の任意の二つの間の変換に対しアルゴリズムが存在するが，我々は，変換の可能性に対してのみ興味を示すのではなく，変換に要する時間についても注意を払うことにする．とりわけ,（入力列の長さに関する関数として）指数時間を必要とするアルゴリズム，したがって，実際には小さいサイズの入力に対してしか実行することのできないアルゴリズムと，入力サイズに関する 1 次，2 次，またはその他の低次の多項式程度の時間で実行可能なアルゴリズムとを，明確に区別することが肝要である．後者のアルゴリズムは，大きな入力に対しても実行可能であることが期待されるという意味で「現実的」なアルゴリズムである．以下で取り上げる各々の変換に対しても，時間計算量を合わせて考察していくことにする．

NFA から DFA への変換　NFA または ε-NFA から出発して，それを DFA に変換するとき，変換に必要な時間計算量は NFA または ε-NFA の状態数に関する指数時間になる場合がある．最初に，n 状態の ε-NFA の ε-閉包の計算を行う．n 個の状態の各々からラベルが ε であるすべての有向辺に沿って探索を行うと $O(n^3)$ 時間を要する．なお，状態数が n であるとき，有向辺の個数が n^2 を超えることはない．データ管理を工夫し，上手に設計されたデータ構造を用いれば，$O(n^2)$ 時間で各状態からの探索を行うことができる．実際，ワーシャル (Warshall) のアルゴリズムのような推移閉包アルゴリズムを用いれば，ε-閉

包の全体を一度に計算してしまうことも可能である[3])．

ひとたび ε-閉包が計算できれば，次は，部分集合構成法を用いて等価な DFA を求めることが可能となる．主要なコストは，原理的に，DFA の状態数であり，それは最大で 2^n 程度になり得る．各状態に対し，入力文字の各々に対する ε-閉包の情報と NFA の遷移表とを参照することにより，$O(n^3)$ 時間ですべての遷移を決定することができる．もう少し詳しく見るため，$\delta(\{q_1, q_2, \ldots, q_k\}, a)$ を計算する場合を考えてみよう．各々の q_i から ε のラベルを持つ路に沿って到達可能な状態は最大で n 個あり得る．そして，それらの状態の各々が a のラベルを持つ有向辺を最大で n 個持ち得る．状態で番号付けされた配列を用意することにより，高々n 個の状態を持つ，高々n 個の集合の集合和を n^2 に比例する時間で求めることができる．

このようにして，各状態 q_i に対し，q_i から a（ε の可能性もある）のラベルを持つ路に沿って到達可能な状態の集合を求めることができる．$k \leq n$ であるので，処理しなければならない状態の個数は高々n 個である．各状態ごとに，到達可能な状態の集合を $O(n^2)$ 時間で計算することができる．したがって，到達可能な状態の集合を計算するために費やされる時間の合計は $O(n^3)$ である．さらに，到達可能な状態の集合の集合和を求めるには $O(n^2)$ 時間を追加するだけでよい．以上より，DFA の一つの遷移を求める計算は $O(n^3)$ 時間で実行できることが結論される．

入力文字の個数は定数である，すなわち，n には依存しないと仮定されていることに注意されたい．したがって，ここでも，またこれ以後も，計算時間の評価を行う際に入力文字の個数を考慮に入れることはしない．入力アルファベットの大きさは，定数係数には影響を与えるが，それは O-記法の中に隠されてしまう．入力アルファベットの大きさがそれ以上の影響を与えることはない．

結論をまとめると，NFA から DFA への変換のための実行時間は，NFA が ε-動作を持つ場合も含めて，$O(n^3 2^n)$ である．もちろん，実際には，作り出される状態数が 2^n よりはるかに少ないという場合も少なくない．n 状態ですんで

---

3) 推移閉包アルゴリズムに関する議論については，エイホ (A. V. Aho)，ホップクロフト (J. E. Hopcroft)，ウルマン (J. D. Ullman) 著，*Data Structures and Algorithms* (Addison-Wesley，1984) ＜**邦訳文献**：大野義夫訳 データ構造とアルゴリズム (培風館，1986) ＞を見よ．

しまう場合もしばしばである．そこで，s を DFA が実際に有する状態数とし，実行時間の上界として $O(n^3s)$ を採用してもよい．

DFA から NFA への変換 この変換は容易である．n-状態の DFA から NFA への変換は $O(n)$ 時間で行うことができる．変換に必要な作業は，DFA に対する遷移表を，遷移先を表す各状態を集合を表す括弧（$\{$ と $\}$）で囲むように書き直すことと，変換の出力が ε-NFA の場合には ε に対応する列を一つ付け加えることだけである．入力文字の個数（すなわち，遷移表の幅）は定数として扱うので，遷移表のコピーと遷移表に対する上記の処理は $O(n)$ 時間で行うことができる．

オートマトンから正則表現への変換 3.2.1 項に述べた構成について調べてみよう．与えられた DFA の状態数を n とするとき，n 回の繰り返しの各回ごとに構成される正則表現の大きさは 4 倍になり得る．前の回に得られた 4 つの表現から作られるからである．したがって，n^3 個の表現を単に書き下すだけで $O(n^3 4^n)$ 時間が必要となる．3.2.2 項にこれを改良した構成法を述べたが，その場合でも，定数係数が減少するだけで，最悪の場合に指数時間を要することに変わりはない．

我々は説明を省いたが，変換される対象が NFA であるとき，さらには ε-NFA であるときでも，同じ構成法を用いて同じ計算時間で変換することができる．ただし，その場合，肝要なのは，直接 NFA に対して構成を行うことである．最初に NFA を DFA に変換し，その後で，得られた DFA を正則表現に変換するという手順をとると，計算時間は $O(8^n 4^{2^n})$ になってしまう．これは，2 重指数時間である．

正則表現からオートマトンへの変換 正則表現から ε-NFA への変換は線型時間で行うことができる．正則表現を効率よく構文解析することが必要であるが，長さ n の正則表現に対し $O(n)$ 時間しか必要としないうまい解析技法があるので，それを用いればよい[4]．解析の結果として得られるのは，正則表現の

---

[4] この仕事を $O(n)$ 時間で行うことのできる構文解析の方法については，エイホ (A. V. Aho)，セシィ(R. Sethi)，ウルマン (J. D. Ullman) 著，*Compilers: Principles, Techniques, and Tools* (Addison-Wesley, 1986) ＜**邦訳文献**：原田賢一訳 コンパイラ I, II 原理・技法・ツール (サイエンス社, 1990) ＞の中で紹介されている．

中の各文字に対し一つの頂点が対応する表現木である.（ただし，括弧を木の中に出現させる必要はない．それらは正則表現の解析の順序を誘導するだけで十分役目を果たしている.）

正則表現に対する表現木が得られたら，その木の各頂点に対して ε-NFA を構成していく．3.2.3 項で正則表現の変換のための構成規則を調べたが，その規則に従えば，表現木の各頂点ごとに高々二つの状態と四つの有向辺が付加されるだけである．したがって，このようにして得られる ε-NFA の状態の個数と有向辺の個数はどちらも $O(n)$ である．さらに，各部分木を処理する関数はポインタをオートマトンの初期状態と受理状態に返すという条件のもとで，解析木の各頂点でこれらの要素を作り出す際に行われる仕事に要する時間は定数時間である．

結論として，正則表現からの ε-NFA の構成は，正則表現の長さに関する線型の時間で実行することができる．さらに，n-状態の ε-NFA から ε-動作を除去して通常の NFA を作る作業は，状態数を増やすことなく，$O(n^3)$ 時間で行うことができる．ただし，そこからさらに DFA の構成にまで進むとなると，指数時間を必要とする場合が起こり得る．

4.3.2　正則言語の空言語判定

「正則言語 L は空言語か」という質問に対する答は，一見しただけでは，自明なことのように思えるかもしれない．つまり,「$\varnothing$ は空言語である．それ以外はすべて空言語ではない.」と答えればそれですむように思えるからである．しかし，4.3 節の冒頭に述べたように，質問は，L に属するすべての文字列のリストを具体的に提示した上で問われる訳ではない．そうではなく，普通は，L に対応する表現のいずれかが与えられ，その表現が言語 $\varnothing$ を表す表現であるかどうかを判定することが求められる．

対象となる表現がいずれかの種類のオートマトンの場合は，空言語問題は，初期状態から受理状態の一つへの路，どんな路でも構わないが，の存在を問う問題になる．そのような路が一つでも存在すれば言語は空言語ではないし，逆に，受理状態がすべて初期状態から切り離されているときは言語は空言語である．初期状態から受理状態へ到達可能かどうかを判定する問題はグラフ到達可能性問題の簡単な一例であり，考え方の上で，2.5.3 項で論じた ε-閉包の計算に

相通じる所がある．以下，アルゴリズムを再帰的に記述する．

基礎：初期状態は，明らかに，初期状態から到達可能である．

再帰：状態 q が初期状態から到達可能であり，q から状態 p に向かう有向辺が存在するとき，p は初期状態から到達可能である．なお，q から p への有向辺のラベルは任意（入力文字，または，ε-NFA の場合は ε）でよい．

このアルゴリズムを用いて，到達可能な状態の集合を求めることができる．もしその中に一つでも受理状態が含まれていれば，我々の答は “no” である（すなわち，オートマトンが受理する言語は空言語ではない）．一方，到達可能な状態の集合の中に受理状態が一つも含まれていなければ，答は “yes” である．到達可能性を調べる計算は，オートマトンの状態数が n であるとき，$O(n^2)$ 時間以下で行えることに注意されたい．実際，最悪でもオートマトンの状態遷移図に含まれる有向辺の個数に比例する程度であり，それは n^2 より小さいことはあり得るが，$O(n^2)$ を超えることはない．

オートマトンではなく，言語 L を表す正則表現が与えられた場合も，正則表現を ε-NFA に変換し，上記の手順に進むことができる．長さ n の正則表現から得られるオートマトンの状態と遷移の個数は高々$O(n)$ である．したがって，このアルゴリズムの実行に要する時間は $O(n)$ 時間である．

一方，正則表現が表す言語が空言語であるかどうかを判定する方法として，上の方法の他に，正則表現から直接調べる方法もある．まず，正則表現に $\varnothing$ が一つも出現していなければ，その正則表現が表す言語は明らかに空言語ではない．正則表現に $\varnothing$ が含まれている場合は，その正則表現が表す言語は空言語である場合もあるし，空言語でない場合もある．次に記す再帰的規則により，与えられた正則表現が表す言語が空言語であるかどうかを判定することができる．

基礎：$\varnothing$ が表す言語は空言語である．ε が表す言語，および，任意の入力文字 a に対し $\mathbf{a}$ が表す言語は空言語ではない．

再帰：R を正則表現とする．R の構成のしかたにより四つの場合に分けられる．

(1) $R = R_1 + R_2$ の場合：$L(R_1)$ と $L(R_2)$ がともに空言語であるとき，またそのときに限り，$L(R)$ は空言語である．

(2) $R = R_1R_2$ の場合：$L(R_1)$ と $L(R_2)$ の（少なくとも）一方が空言語であるとき，またそのときに限り，$L(R)$ は空言語である．

(3) $R = R_1^*$ の場合：$L(R)$ は空言語ではない．$L(R)$ は常に ε を含むから

である．

(4)　$R = (R_1)$ の場合：$L(R_1)$ が空言語であるとき，またそのときに限り，$L(R)$ は空言語である．これらの二つは同じ言語であるからである．

4.3.3　正則言語における所属性判定

次に取り上げる重要な問題は，所属性判定問題と呼ばれる，次の問題である．文字列 w と正則言語 L が与えられたとき，w は L に属すかどうかを判定せよ．なお，w は文字列そのものが具体的に与えられるが，L はオートマトンによって，または，正則表現によって表現されるとする．

L が DFA によって表現されている場合は，所属性を判定するアルゴリズムは簡単である．初期状態で，w を入力列として受け取るところから処理をはじめ，DFA の動作を模倣していけばよい．処理が終了したとき DFA が受理状態にあれば，答は "yes" であり，そうでなければ，答は "no" である．このアルゴリズムの計算時間はとても速い．$|w| = n$ とし，DFA の状態遷移表が 2 次元配列のような適切なデータ構造を用いて表現されているとすれば，1 回の遷移には定数時間しかかからず，判定の処理全体は $O(n)$ 時間を要するだけである．

次に，L が DFA 以外のものによって表現されている場合は，それをまず DFA に変換し，しかる後に，上記の判定を行うのが一つの方法である．この方法は，表現の長さに関して計ると指数時間を要することがあり得るが，$|w|$ に関して計れば線型時間である．しかし，表現が NFA や ε-NFA である場合には，直接 NFA の動作を模倣する方法による方が簡単であるし，効率もよい．すなわち，w に含まれる文字を一つずつ順に見ていき，その時点までの w の接頭語でラベルづけされた路を通って取り得る状態の集合を保持しながら，処理を進めていけばよい．この考え方は図 2.10 に紹介されている．

このアルゴリズムの実行時間は，w の長さを n とし，NFA の状態数を s とするとき，$O(ns^2)$ である．入力文字の一つ一つの処理は，その時点の直前に得られた状態の集合（高々 s 個の状態の集合）を取り出し，これらの状態の各々についてその次に来ることのできる状態をすべて求めていくことによって行えばよい．各々が高々 s 個の状態から成る，高々 s 個の集合の和集合を作ることになるが，そのために要する時間は $O(s^2)$ 時間である．

NFA が ε-動作を持つ場合は，NFA の模倣を始める前に ε-閉包の計算をしな

ければならない．このとき，各入力文字 a の処理には二段階の作業が必要であり，それぞれが $O(s^2)$ 時間を要する．第一の作業は，その時点の直前に得られた状態の集合について，そこに含まれる各状態が入力文字 a によって次にとることのできる状態をすべて求めることである．次の作業は，この状態の集合について ε-閉包を計算することである．模倣を開始する時点で対象とする最初の状態の集合は，初期状態の ε-閉包である．

最後に，L が長さ s の正則表現で表現されている場合を考える．このときは，まず，正則表現を高々$2s$ 個の状態から成る ε-NFA に変換する．このために要する時間は $O(s)$ 時間である．次に，上に述べた模倣を行う．このために要する時間は，長さ n の入力 w に対し，$O(ns^2)$ 時間である．

4.3.4　4.3 節の演習問題

♣ * **問 4.3.1**　正則言語 L が無限集合であるかどうかを判定するアルゴリズムを与えよ．ヒント：反復補題を用いて，ある下限値を超える長さの文字列を含む言語は無限集合であることを示せ．

♣ **問 4.3.2**　正則言語 L が 100 個以上の文字列を含むかどうかを判定するアルゴリズムを与えよ．

♣ **問 4.3.3**　L をアルファベット Σ 上の正則言語とする．$L = \Sigma^*$ であるかどうか，すなわち，L がアルファベット Σ 上のすべての文字列から成る言語であるかどうかを判定するアルゴリズムを与えよ．

♣ **問 4.3.4**　二つの正則言語 L_1 と L_2 が少なくとも一つの文字列を共有するかどうかを判定するアルゴリズムを与えよ．

♣ **問 4.3.5**　同じアルファベット Σ 上の 二つの正則言語 L_1 と L_2 に対し，Σ 上の文字列で L_1 にも L_2 にも属さないものが存在するかどうかを判定するアルゴリズムを与えよ．

4.4　オートマトンの等価性と最小性

前節で考察した問題――空言語と所属性――には簡単なアルゴリズムが存在したが，それとは対照的に，二つの正則言語を表す二つの記述が，実は，同じ言語を定義しているかどうかを判定する問題では，相当に深い知的操作が要求される．本節では，正則言語に対する二つの記述が，それらが同じ言語を定義するという意味で，**等価** (equivalent) であるかどうかを判定する方法について論じる．この判定法の一つの重要な帰結として，DFA を最小化する方法が存在

することが示される．すなわち，我々は，任意の DFA に対し，それと等価な DFA で状態の個数が最小のものを見つけることができる．実は，この状態数最小の DFA は本質的にただ一つしか存在しない．もう少し詳しくいうと，互いに等価である二つの最小状態数の DFA が与えられたとき，それらの状態の名前をつけかえることにより二つの DFA を同じにすることが常に可能である．

4.4.1　状態の同値性の判定

まず，一つの DFA における二つの状態に関する問題を考えることから始めよう．目標は，二つの相異なる状態 p と q を，p と q の両方と同様に振る舞う一つの状態によって置き換えることができるのはどんな場合であるかを理解することである．状態 p と q が次を満たすとき，p と q は**同値** (equivalent) であるという．

- すべての入力列 w に対し，$\hat{\delta}(p,w)$ が受理状態であるための必要十分条件は $\hat{\delta}(q,w)$ が受理状態であることである．

直観的にいうと，二つの状態 p と q が同値であるとは，これらのうちの一方を初期状態として与えられた入力列を読み進むとき受理状態に到達するかどうかを調べるだけでは，両者の間を区別することは不可能であるということを意味する．我々は，必ずしも $\hat{\delta}(p,w)$ と $\hat{\delta}(q,w)$ とが**同じ**状態であることを要求しているわけでは**ない**ことに注意されたい．両者がともに受理状態であるか，または，両者がともに受理状態ではないならば，それでよいのである．

二つの状態が同値でないとき，それらは**区別可能** (distinguishable) であるという．すなわち，状態 p が状態 q から区別可能であるのは，$\hat{\delta}(p,w)$ と $\hat{\delta}(q,w)$ のうち一方は受理状態であるが，他方は受理状態ではない，となるような文字列 w が少なくとも一つ存在することである．

例 4.18　図 4.8 の DFA を考えてみよう．ここでは，状態遷移関数を δ と表す．状態の対のうちのいくつかは，明らかに，同値ではない．たとえば，C と G は同値ではない．一方は受理状態であるのに，もう一方は受理状態ではないからである．すなわち，$\hat{\delta}(C,\varepsilon)$ は受理状態であるが，$\hat{\delta}(G,\varepsilon)$ は受理状態ではないので，空列 ε によってこれらの二つの状態は区別される．

次に，二つの状態 A と G を考えてみよう．今度は，空列 ε によってこれらの二つの状態が区別されることはない．両者はいずれも受理状態ではないからで

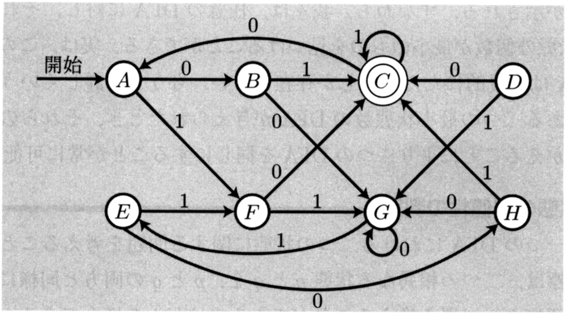


図 4.8　同値な状態を持つオートマトン

ある．また，文字列 0 も状態 A と G を区別することはない．文字列 0 に対し A と G は，それぞれ，B と G に移るが，これらはいずれも受理状態ではないからである．同様に，文字列 1 も状態 A と G を区別することはない．文字列 1 に対し A と G は，それぞれ，F と E に移るが，いずれも受理状態ではないからである．しかし，文字列 01 によって A は G から区別される．$\hat{\delta}(A, 01) = C$, $\hat{\delta}(G, 01) = E$ であるが，C は受理状態であるのに，E は受理状態ではないからである．A と G のうち一方を受理状態へ導き，他方を受理状態ではない状態へ導くような文字列が一つでも存在することが確かめられれば，それだけで，A と G が同値ではないことを示すのに十分である．

今度は，二つの状態 A と E を考えてみよう．どちらも受理状態ではないので，ε によってこれらの状態が区別されることはない．入力 1 によりこれらはいずれも状態 F に移る．したがって，任意の文字列 x に対し $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x)$ となるので，1 で始まる入力列によって A と E を区別することはできない．

続いて，0 で始まる入力列が与えられたとき，二つの状態 A と E からの動きがどうなるかを考えてみる．A と E はまず，入力 0 によって，それぞれ，状態 B と H とに移る．これらはいずれも受理状態ではないので，文字列 0 によって A と E とを区別することはできない．ところが，B と H は状態を区別するという観点からは全く無力であることがわかる．入力 1 で両者はともに C に移り，また，入力 0 で両者はともに G に移るからである．よって，0 で始まる入力列のどれによっても A と E を区別することは不可能である．以上より，いか

なる入力列も A と E を区別することはできないと結論することができる．すなわち，この二つの状態は同値である．▨

同値な状態をすべて見つけるには，逆に，区別可能な状態の対を可能な限り見つけ出す作業を行うとよい．以下にアルゴリズムを記すが，このアルゴリズムに従う作業を可能な限り続けるとき，最後まで区別可能であると判定されずに終わる状態の対はすべて同値である．これは信じがたいことのように思えるかもしれないが，事実である．次のアルゴリズムを**（表の）穴埋めアルゴリズム** (table-filling algorithm) と呼ぶが，これは，DFA $A = (Q, \Sigma, \delta, q_0, F)$ における区別可能な状態の対を再帰的に発見していくアルゴリズムである．

基礎：状態 p が受理状態であり，状態 q が受理状態でないならば，対 $\{p, q\}$ は区別可能である．

再帰：状態 p と q は，ある入力文字 a に対し $r = \delta(p, a), s = \delta(q, a)$ となり，$\{r, s\}$ は区別可能であることがすでに確かめられている状態の対であるとする．このとき，$\{p, q\}$ は区別可能な状態の対である．この判定が正しいのは以下の理由による．まず，r と s を区別する文字列 w が必ず存在する．すなわち，$\hat{\delta}(r, w)$ と $\hat{\delta}(s, w)$ のうち一方のみが受理状態である．そうすると，文字列 aw は p と q とを区別する．$\{\hat{\delta}(p, aw), \hat{\delta}(q, aw)\}$ は $\{\hat{\delta}(r, w), \hat{\delta}(s, w)\}$ と同じ状態の対になるからである．

例 4.19　図 4.8 に示した DFA に対し，穴埋めアルゴリズムを実行してみよう．最終結果を図 4.9 に示す．ここで，x は区別可能な状態の対を表し，空白は同値であると判定された状態の対を表す．

最初，表の中に x は一つも記入されていない．

C がただ一つの受理状態であるので，アルゴリズムの基礎として，C を含むすべての対に x を記入する．さて，区別可能な対がいくつか見つかったので，これを出発点として，さらに他の区別可能な対を見つけていく．たとえば，$\{C, H\}$ は区別可能であり，E と F は入力 0 が与えられると，それぞれ，H と C に行くので，$\{E, F\}$ もやはり区別可能な対である．実際，図 4.9 の中の x のうち対 $\{A, G\}$ と対 $\{E, G\}$ に対する x を除くすべての x は，単にある状態の対からの 0 または 1 に対する遷移を見て，これらの入力の一つに対し一方の状態は C に行き，他方は C に行かないことを確かめるだけで決定することができる．$\{A, G\}$

	A	B	C	D	E	F	G
B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x

図 4.9　状態の非同値性を示す表

と $\{E, G\}$ とがそれぞれ区別可能であることは，再帰的手順の次の回の適用の際に発見することができる．入力 1 に対し，A と E は F に行き，G は E に行くが，対 $\{E, F\}$ が区別可能であることはすでに前の回にわかっているからである．

さて，ここまで進んだ段階で，我々は，これらの他にさらに区別可能な対を発見することはできない．残った三つの対は $\{A, E\}$，$\{B, H\}$，$\{D, F\}$ であるが，これらはいずれも同値な対である．一例として，対 $\{A, E\}$ はなぜ区別可能な対ではないのかを考えてみよう．入力 0 に対し，A と E は，それぞれ，B と H に行く．対 $\{B, H\}$ はまだ区別可能であることが示されてはいない．また，入力 1 に対し，A と E はどちらも同じ状態 F に行く．したがって，それ以上進めても A と E が区別される望みは全くない．他の二つの対 $\{B, H\}$ と $\{D, F\}$ はいずれも，対をなす二つの状態の 0 に対する遷移先と 1 に対する遷移先がそれぞれ同じ状態になっている．したがって，穴埋めアルゴリズムは，図 4.9 に示す表が得られたところで停止する．そして，その結果は，同値な状態の対と区別可能な状態の対の正しい決定となっている．▨

定理 4.20　二つの状態が穴埋めアルゴリズムによって区別されないならば，それらの状態は同値である．

証明　考察の対象である DFA を $A = (Q, \Sigma, \delta, q_0, F)$ とする．定理は誤りであると仮定する．したがって，次のような状態の対 $\{p, q\}$ が少なくとも一つ存在する．

(1)　状態 p と q は区別可能である．すなわち，$\hat{\delta}(p, w)$ と $\hat{\delta}(q, w)$ のうち一方だけが受理状態であるようなある文字列 w が存在する．

(2) 穴埋めアルゴリズムによって p と q は区別されない.

このような状態の対を**悪い対** (bad pair) と呼ぶことにする.

悪い対が一つでも存在する場合には, 悪い対（のいずれか）を区別する文字列の全体の中で最小の長さの文字列によって区別される対がある. そのような悪い対の一つを $\{p,q\}$ とし, $w = a_1a_2\cdots a_n$ を p と q を区別する文字列の中で長さが最小の文字列であるとする. したがって, $\hat{\delta}(p,w)$ と $\hat{\delta}(q,w)$ のうち一方だけが受理状態である.

まず最初に, w は ε ではあり得ないことに注意する. 一般に, ε がある状態の対を区別するとすれば, 穴埋めアルゴリズムの基礎の部分で, その対には（区別可能であることを示す）印 x が付されるからである. そこで以下, $n \geq 1$ とする.

二つの状態 $r = \delta(p,a_1)$ と $s = \delta(q,a_1)$ を考えよう. r と s は文字列 $a_2a_3\cdots a_n$ によって区別される. この文字列により r と s は, それぞれ, $\hat{\delta}(p,w)$ と $\hat{\delta}(q,w)$ に移るからである. ところが, r と s を区別するこの文字列は, 悪い対を区別するどの文字列よりも短い文字列である. したがって, $\{r,s\}$ は悪い対ではあり得ない. そこで, 穴埋めアルゴリズムによってこれらは区別可能であると判定されるはずである.

しかし, $\delta(p,a_1) = r$ と $\delta(q,a_1) = s$ が区別可能であることが発見される以上, p と q も区別可能であることを発見する以前に, 穴埋めアルゴリズムの再帰的部分が実行を停止することはあり得ない. 以上により, 我々は, 悪い対が存在するという仮定に対し矛盾する結果を得た. 悪い対が一つも存在しないということは, 区別可能な対はすべて必ず穴埋めアルゴリズムによって区別されるということであり, 定理が正しいことが証明された. ▨

4.4.2 正則言語の等価性の判定

穴埋めアルゴリズムを用いることにより, 二つの正則言語が同じ言語であるかどうかを判定する簡単な方法が得られる. 言語 L と M が, それぞれ適当な表現によって与えられているとする. たとえば, 一方は正則表現によって与えられ, もう一方は NFA によって与えられるというようにである. まず, それぞれの表現を DFA に変換する. さてその上で, L を受理する DFA の状態集合と M を受理する DFA の状態集合の集合和を状態集合として持つ, もう一つの DFA を想像していただきたい. 定義上はこの DFA は初期状態を二つ持つことになるが, 状態の同値性の判定が考察の対象となっている限りでは, 初期状態の役割は不要である. 任意の状態を（ただ一つの）初期状態として定めておいてさしつかえない.

さて, 二つの元々の DFA で定義されていた初期状態同士について, それらが同値であるかどうかを, 穴埋めアルゴリズムを用いて調べる. それらが同値

であれば $L = M$ であり，同値でなければ $L \neq M$ である．

例 4.21　図 4.10 に示す二つの DFA について考えてみよう．どちらの DFA も，末尾が 0 のすべての文字列と空列を受理する．すなわち，受理する言語は正則表現 $\varepsilon + (\mathbf{0} + \mathbf{1})^*\mathbf{0}$ に対応する言語である．ここで我々は，図 4.10 を，二つのオートマトンではなく，A から E までの 5 個の状態を持つ一つの DFA を表すものと考える．そう考えたオートマトンに対して穴埋めアルゴリズムを適用すると，図 4.11 に示すような結果が得られる．

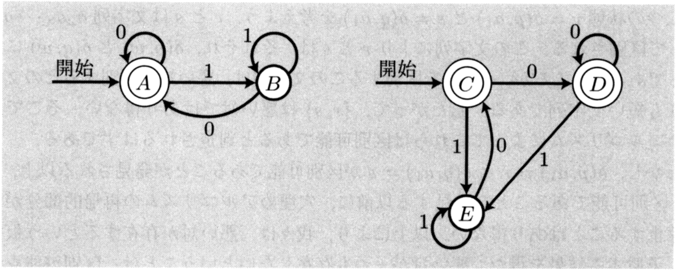


図 4.10　二つの互いに等価な DFA

	A	B	C	D
B	x			
C		x		
D		x		
E	x		x	x

図 4.11　図 4.10 に対する区別可能性の表

この表に x が記入されていく手順を見てみよう．最初の作業は，一方だけが受理状態であるような状態の対に対応するすべての位置に x を書き込むことである．ところが，この例の場合，書き込みの作業はこれでおしまいである．残った四つの対 $\{A, C\}$, $\{A, D\}$, $\{C, D\}$, $\{B, E\}$ はすべて同値な対である．読者は，穴埋めアルゴリズムの再帰的な部分においてこれ以上区別可能な対は発見されないことを確かめていただきたい．たとえば，図 4.11 のような表の状況

が与えられているとき，対 $\{A, D\}$ を区別することはできない．入力 0 に対してそれらはいずれも自分自身に移るし，入力 1 に対しては，まだ区別されていない対 $\{B, E\}$ に移るからである．最終的な判定結果より，A と C は同値であることがわかる．これらの状態は元の二つのオートマトンの初期状態であった．よって，元の二つの DFA は同じ言語を受理することが結論される．▨

表の書き込みを完成するのに要する時間，したがって，二つの状態が同値であるかどうかを判定するための時間は，状態の個数に関する多項式程度の時間である．状態数が n の場合，状態の対の個数は $\binom{n}{2}$，すなわち，$\dfrac{n(n-1)}{2}$ である．繰り返しの処理の 1 ラウンドごとに，状態の対のすべてについて，それらの 1 文字による移り先の対の中にすでに区別可能と判定された対があるかどうかを調べる．このために必要な時間は $O(n^2)$ 時間を超えることはない．また，1 回の繰り返しの間に表の中に新たに x が記入されることがなければ，その時点でアルゴリズムの実行は終了する．したがって，繰り返しの回数が $O(n^2)$ を超えることはなく，全体として，$O(n^4)$ が穴埋めアルゴリズムの実行時間の一つの上界となる．

ただし，実は，もう少し注意深くアルゴリズムを適用すると $O(n^2)$ 時間で表の書き込みを完了することができる．そのためのアイデアは，状態の対 $\{r, s\}$ の各々について，$\{r, s\}$ に「依存する」状態の対 $\{p, q\}$ のリストをあらかじめ初期設定しておくことにある．ここで，$\{p, q\}$ が $\{r, s\}$ に依存するとは，$\{r, s\}$ が区別可能であることがわかれば，そのことから直接 $\{p, q\}$ も区別可能であることがわかることをいう．リストの初期設定は次のように行う．状態の対 $\{p, q\}$ の各々と，定数個の入力文字 a の各々について，対 $\{p, q\}$ を，入力 a に対する p, q の移り先である対 $\{\delta(p, a), \delta(q, a)\}$ に対するリストの中に挿入する．

対 $\{r, s\}$ が区別可能であることがわかったら，$\{r, s\}$ に対するリストを逐次たどっていく．そして，そのリストの中でまだ区別可能とされていないすべての対に対し，まず，その対に区別可能の印をつけ，次に，その対を，チェック待ちのリストを持つ対のキューに押し込む．

このアルゴリズムの実行に必要な時間の全体は，各対のリストの長さの和に比例する時間である．我々が行う作業は，リストに何かを付け加える作業（初

期設定）か，またはリストに属する要素を調べる作業（これは，区別可能であることがわかった対に対するリストをたどっていくときに行われる作業であり，どの要素についても 2 回以上行われることはない）のいずれかだけだからである．入力アルファベットの大きさは定数とみなすので，状態の対の各々は $O(1)$ 個のリストに挿入されるだけである．対の総数は $O(n^2)$ 個であることより，全体の実行時間は $O(n^2)$ となる．

4.4.3 DFA の最小化

状態の同値性を判定するテストからは，もう一つの重要な結論が導かれる．それは，DFA を「最小化する」ことが可能だということである．すなわち，任意の DFA に対し，それと同じ言語を受理する DFA の中で状態数が最小の DFA を見つけることができる．そればかりでなく，任意の正則言語に対し，状態につける名前の選び方の自由度を除いて，状態数最小の DFA はただ一つしか存在しないこともわかる．

DFA の最小化を支える中心的なアイデアは，状態の同値という概念により，状態の全体を次の条件を満たすいくつかのブロックに分割することができる，というものである．

(1) 一つのブロックに属す状態は，すべて互いに同値である．

(2) 相異なるブロックに属す二つの状態は同値ではない．

例 4.22 図 4.9 の表について考えてみよう．これは，図 4.8 の状態集合に対し，同値な状態の対と区別可能な状態の対とを求めたものであった．同値な状態のブロックへの分割は $(\{A,E\},\{B,H\},\{C\},\{D,F\},\{G\})$ である．同値な状態の対は三組ある．それらは，各々，同じブロックに配置される．一方，自分自身以外のすべての状態と区別可能な状態はそれぞれ一つの状態だけからなるブロックを構成する．

図 4.10 のオートマトンでは，分割は $(\{A,C,D\},\{B,E\})$ である．この例から，一つのブロックの中に 3 個以上の状態が属す場合があることがわかる．A,C,D の三つの状態は一つのブロックの中に共存している．それは，これらのうちの二つはいずれも互いに同値であり，また，これらのうちのどれもこれら以外の状態とはどれとも同値ではないからである．このような三つの状態の共存は，あるいは偶発的なことのように思えるかもしれない．しかし，次に証

明する定理に見るように，このような状況は，状態に対する「同値」の定義によって保証されることなのである． ▨

定理 4.23 状態集合の上で定義された同値という関係は推移律を満たす．すなわち，任意の DFA $A = (Q, \Sigma, \delta, q_0, F)$ に対し，p, q, r を三つの状態とするとき，p と q が同値であり，かつ，q と r も同値であるならば，p と r もまた同値である．

証明 推移律は，一般に,「同値関係」と呼ばれる関係において常に成り立つ性質であることに注意する．ただし，何らかの対象について，単にそれを「同値」と呼んだからといって，ただそれだけでその対象が推移的になるわけではもちろんない．我々は，同値と呼んだことの正当性を証明しなければならない．

二つの対 $\{p,q\}$ と $\{q,r\}$ はそれぞれ同値であるが，対 $\{p,r\}$ は同値ではないと仮定する．そうすると，$\hat{\delta}(p,w)$ と $\hat{\delta}(r,w)$ のうち一方だけが受理状態となるような入力列 w が存在する．対称性により一般性を失うことなく，$\hat{\delta}(p,w)$ は受理状態であるとしてよい．

さて，このとき $\hat{\delta}(q,w)$ は受理状態であるかどうかを考えてみる．もし $\hat{\delta}(q,w)$ が受理状態であるとすると，$\hat{\delta}(q,w)$ は受理状態であり，$\hat{\delta}(r,w)$ は受理状態ではないことから，$\{q,r\}$ は区別可能となる．一方，$\hat{\delta}(q,w)$ が受理状態ではないとすると，同様の理由により，今度は $\{p,q\}$ が区別可能となる．いずれにしても矛盾が生じる．よって，$\{p,r\}$ は区別可能ではなく，したがって，同値である． ▨

定理 4.23 を用いると，状態の集合を分割するための一つのわかりやすいアルゴリズムについて，その正当性を証明することができる．各状態 q に対し，q と，q に同値なすべての状態とからなるブロックを構成する．こうして得られるブロックの全体は分割である．すなわち，どの状態も二つの異なるブロックに属すことはない．このことを以下に示す．

まず最初に，同じブロックに属す状態はすべて互いに同値であることに注意する．なぜならば，p と r がいずれも q に同値な状態から成るブロックに属しているとすると，定理 4.23 より，p と r は互いに同値になるからである．

さて，全く同じではないが，重なる部分を持つ二つのブロックが存在するとする．すなわち，p と q を含むブロック B と，p は含むが q は含まないブロック C があるとする．p と q は同じブロックに属すので，それらは同値である．ここで，ブロック C がどのようにして形成されたブロックであるかを考えてみ

る．もし C が p によって生成されたブロックであるとすると，p と q は同値なので q も C に属すことになる．ところが，これは仮定に反する．そこで，C を生成する第3の状態 s が存在し，C は s と同値な状態全体の集合であるとする．p はブロック C に属すので，p は s に同値である．また，p と q は同じブロック B に属すので，p は q に同値である．したがって，定理 4.23 の推移性により，q は s に同値となる．しかしそうなると，q は C に属すことになり，矛盾が生じる．

以上より，状態の間の同値関係により状態集合の分割が得られることが結論される．すなわち，任意の二つの状態に対し，それらの（自分自身も含めた）同値な状態の集合を考えると，それらは全く同じ集合であるか，または，共通部分を全く持たないかのいずれかである．いま述べた考察の結果をまとめて次の定理を得る．

定理 4.24　DFA の各状態 q に対し，q と，q に同値なすべての状態とから成る**ブロック**（状態集合の部分集合）を構成する．このとき，相異なるブロックの全体は状態集合の**分割**を与える[5]．すなわち，どの状態もただ一つのブロックに属す．同じブロックに属す状態はすべて互いに同値であり，異なるブロックに属す状態同士は同値ではない．　▨

さて，アルゴリズムを簡潔に記述するための準備が整ったので，次に，DFA $A = (Q, \Sigma, \delta, q_0, F)$ を最小化するためのアルゴリズムを以下に記す．

(1)　穴埋めアルゴリズムを用いて，互いに同値な状態の対をすべて求める．

(2)　上に述べた手順により，状態の集合 Q を，互いに同値な状態の集合からなるいくつかのブロックに分割する．

(3)　各ブロックを一つの状態と考えることにより，DFA A と等価な DFA の中で状態数が最小の DFA B を構成する．B の状態遷移関数を γ とする．S を A の一つの同値な状態の集合とし，a を入力文字とする．このとき，状態のブロック T で，S に属すすべての状態 q に対し $\delta(q, a)$ はブロック T の要素となる，という性質を満たすものが必ず存在する．なぜ

---

[5] 同じブロックが，相異なる状態を生成元として，何度も重複して形成されることに留意されたい．ただし，分割は**相異なる**ブロックによって構成されるので，一つのブロックが重複して形成されるとしても，分割の中にそのブロックはただ一度出現するだけである．

ならば，もしこのようなブロック T が存在しないとすると，S の二つの状態 p と q で入力文字 a によって互いに異なるブロックに属す二つの状態に移るものが存在することになる．定理 4.24 により，これらの（移り先の）2 状態は区別可能である．そしてこのことから，p と q は同値ではない，したがって，p と q が同じブロック S に属すことはないという結論が導かれてしまう．これは仮定に反する．この結果，$\gamma(S,a) = T$ と定めてよいことがわかる．さらに，次のように定義する．

(a)　B の初期状態は，A の初期状態を含むブロックである．

(b)　B の受理状態の集合は，A の受理状態を含むブロック全体の集合である．各ブロックについて，その中のある一つの状態が受理状態であれば，そのブロックに属すすべての状態が受理状態であることに注意されたい．その理由は，任意の受理状態は，受理状態ではない任意の状態と区別可能であり，したがって，互いに同値な状態から成る一つのブロックの中に受理状態と受理状態ではない状態とが共存することはあり得ないことによる．

例 4.25　図 4.8 の DFA を最小化してみよう．例 4.22 で状態集合の分割を与えるブロックを求めた．それを利用して構成した状態数最小のオートマトンを図 4.12 に示す．オートマトンの五つの状態は，図 4.8 のオートマトンに対する同値な状態の五つのブロックに対応する．

図 4.8 のオートマトンの初期状態は A であるから，状態数最小のオートマトンの初期状態は $\{A,E\}$ である．また，図 4.8 のオートマトンの受理状態は C だけなので，$\{C\}$ は状態数最小のオートマトンのただ一つの受理状態である．図 4.12 の状態遷移は，図 4.8 の状態遷移を正しく反映している．たとえば，図 4.12 では入力 0 に対して状態 $\{A,E\}$ は状態 $\{B,H\}$ へ遷移する．図 4.8 では入力 0 に対して A は B に移り，E は H に移ることから，このことは容易に納得することができる．同様に，入力 1 に対して状態 $\{A,E\}$ は状態 $\{D,F\}$ へ遷移する．図 4.8 を調べると，入力 1 に対して A と E はともに F に移ることがわかる．したがって，$\{A,E\}$ の入力 1 に対する移り先の選択も正しい選択であるといえる．なお，入力 1 に対して A も E も D には移らないが，そのことは何ら差し障りのあることではない．この他の遷移についてもすべて正しい遷

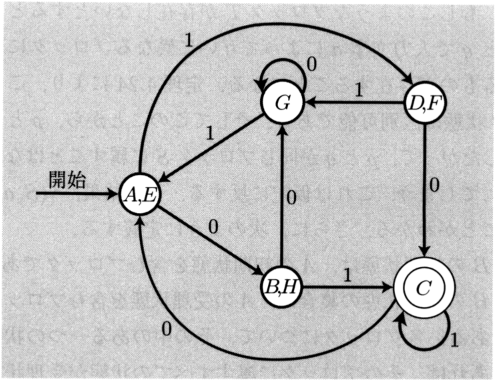


図 4.12 図 4.8 と等価な状態数最小の DFA

移であることを同様に確かめることができる. ▨

4.4.4 最小化 DFA が最小である理由

与えられた DFA A を，定理 4.24 の状態集合分割の方法を用いて最小化し，DFA M を得たとする．定理 4.24 によれば，A の状態集合をさらに少ない個数のグループに分割し，なおかつ，A と等価であるような DFA を作ることはできない．しかし，A とは全く関係のないもう一つの DFA N で，A や M と同じ言語を受理し，しかも，状態数が M より少ないものが存在することはないだろうか．以下，背理法により，このような N は存在しないことを証明しよう．そのため，上のような N が存在すると仮定して議論を進める．

まず，M と N の状態を一緒にして，全体を一つの DFA のようにみなして，4.4.1 項に述べた状態の区別可能性判定の処理を行う．ここで，M の状態と N の状態が同じ名前を持つことはないと仮定してよい．したがって，M と N の状態遷移は互いに独立と考えてよく，二つを合体して得られるオートマトンの状態遷移関数は，M と N のそれぞれの状態遷移規則の和である．合体された DFA の状態が受理状態であるのは，その状態がもとの DFA において受理状態であるときであり，また，そのときに限る．

$L(M) = L(N)$ であるので，M の初期状態と N の初期状態は区別不能であ

NFA に対する状態数の最小化

与えられた NFA または DFA に等価な NFA で状態数が最小のものを求めようとする際にも，DFA の状態数を最小化するために用いられた状態集合を分割する手法と同じ手法を利用することができると思われるかもしれない．しらみつぶし的に数え上げていく作業を行えば，与えられた正則言語を受理する NFA で状態数が最小のものを求めることはできる．しかし，その言語を受理するある与えられた NFA に対し，単にその状態集合をグループ分けするという方法によって求めることはできない．

うまくいかない実例を図 4.13 に示す．この NFA では，三つある状態のうちどの二つも同値ではない．実際，受理状態 B は，受理状態ではない二つの状態 A, C とは区別可能である．また，A と C は入力 0 により区別可能である．C の移り先の集合は A だけからなる集合であり，これは受理状態を含まない．しかし，A の移り先の集合は $\{A, B\}$ であり，これは確かに受理状態を含む．したがって，互いに同値な状態をひとまとめにするグループ分けを行っても，図 4.13 の状態数が減ることはない．

ところが，我々は，同じ言語を受理する NFA でもっと小さいものを見つけることができる．それは簡単で，状態 C を削除してしまえばよい．A と B だけからなる NFA は，末尾が 0 であるようなすべての文字列を受理する．一方，これに状態 C を付け加えても受理される文字列がふえることはない．

る．さらに，$\{p, q\}$ が区別不能な対であるとすれば，任意の 1 個の文字に関するそれらの移り先の二つの状態もまた区別不能である．それは，もし移り先の 2 状態が区別可能ならば，p と q も区別可能になってしまうことから明らかである．

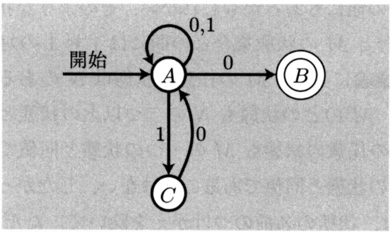


図 4.13　状態の同値を利用する方法では最小化できない NFA の例

M も N も，それぞれの初期状態から到達不能な状態を含むことはない．もしそのような状態が含まれているとすれば，その状態を取り除くことにより，同じ言語を受理する DFA でさらに状態数の少ないものが得られてしまうからである．以上より，M の任意の状態は N の少なくとも一つの状態とは区別不能であることが導かれる．理由は次の通りである．p を M の任意の状態とする．このとき，M の初期状態を状態 p に移すようなある文字列 $a_1a_2 \cdots a_k$ が存在する．この文字列により N の初期状態も N のある状態 q に移る．上に見たように二つの初期状態は区別不能であるので，それらの入力文字 a_1 による移り先も区別不能である．さらに，それらの状態の入力 a_2 による移り先も区別不能である．同様の考察を繰り返すことにより，結局，p と q は区別不能であることが示される．

仮定より N の状態数は M より少ないので，M の二つの状態で N の同じ一つの状態と区別不能であるものが存在する．明らかに，これらの二つの状態は互いに区別不能である．しかし，M は，すべての状態が互いに区別可能であるように作られていたはずである．こうして矛盾が生じたので，N が存在するという仮定が誤りであり，M は，実際，A と等価な DFA の中で状態数が最小の DFA であることが示された．いま証明したことを定理としてまとめておく．

定理 4.26　A を DFA とし，M を，定理 4.24 の直後に記述されたアルゴリズムによって A から構成された DFA とする．このとき，M は A と等価な DFA の中で状態数が最小の DFA である．　▨

実は，定理 4.26 よりもっと強い性質を示すことができる．A と等価で状態数最小の DFA は M の他にもあるかもしれない．そのような状態数最小の任意の DFA N の状態集合と M の状態集合との間には 1 対 1 の対応が存在する．理由は上で展開した議論による．M の任意の状態は N のある状態と同値でなければならず，また，M のどの状態も N の二つ以上の状態と同値であることはない．同様に，N の任意の状態も M の一つの状態と同値でなければならないが，M の二つ以上の状態と同値であることはない．したがって，A と等価な状態数最小の DFA は，状態の名前のつけかえを除いて，ただ一つだけ存在する．

4.4.5　4.4 節の演習問題

♣ * **問 4.4.1**　図 4.14 に，ある DFA の状態遷移表を示す．

a)　このオートマトンに対する区別可能性の表を求めよ．

b)　このオートマトンに等価な DFA で状態数が最小のものを構成せよ．

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

図 4.14　最小化すべき DFA

♣ **問 4.4.2**　図 4.15 の DFA に対し，問題 4.4.1 と同じ質問に答えよ．

	0	1
$\rightarrow A$	B	E
B	C	F
$*C$	D	H
D	E	H
E	F	I
$*F$	G	B
G	H	B
H	I	C
$*I$	A	E

図 4.15　最小化すべきもう一つの DFA

♣ !! **問 4.4.3**　状態数 n の DFA A に対し，A の 二つの状態 p と q は区別可能であるとする．p と q とを区別する文字列のうち最小のものの長さに対するもっとも小さい上界はいくつか．n の関数として答えよ．

4.5 第 4 章の要約

◆**反復補題**：言語が正則であるとき，その言語に属す十分に長い文字列はすべて「ポンプで汲み上げる」ことができる（空ではない）部分列を持つ．すなわち，その部分列を何回繰り返して挿入しても，得られる文字列はやはりその言語に属す．この事実を利用して，いくつもの異なる言語に対し，それらが正則ではないことを証明することができる．

◆**正則言語であるという性質を保存する演算**：正則言語に対して施すと得られる結果が再び正則言語となるような演算が数多く存在する．和，連接，閉包，共通部分，補集合，差，反転，準同型写像（文字の文字列による置き換え），逆準同型写像などがその例である．

◆**正則言語の空集合判定**：オートマトンや正則表現などの正則言語に対する一つの表現が与えられたとき，その表現に対応する言語が空集合であるかどうかを判定するアルゴリズムが存在する．

◆**正則言語に対する所属性判定**：一つの文字列と，正則言語に対する一つの表現とが与えられたとき，その文字列がその言語に属すかどうかを判定するアルゴリズムが存在する．

◆**状態の区別可能性判定**：DFA の二つの状態が区別可能であるとは，その一方を受理状態に導き，もう一方を受理状態ではない状態に導くような入力列が存在することである．受理状態と非受理状態から成る対はすべて区別可能であるという事実だけを出発点とし，一つの入力文字に関する移り先がすでに区別可能であることが示されている状態であるような状態の対を順に見つけ出し，それらを区別可能な状態の対のリストに付け加えることを繰り返す．この操作により，区別可能なすべての対を発見することができる．

◆**決定性有限オートマトンの最小化**：任意の DFA の状態集合を，互いに区別不能な状態から成るいくつかのグループに分割することができる．二つの相異なるグループに属す二つの状態は区別可能である．各グループを一つの状態で置きかえることにより，与えられた DFA と等価な DFA で，同じ言語を受理する DFA の中で状態の個数が最小のものが得られる．

4.6 第 4 章の参考文献

正則表現に対する自明な閉包性——和，連接，スター閉包——は S. C. Kleene [6] によって示されたものであるが，それらを除き，正則言語の閉包性に関するほとんどすべての結果は，文脈自由言語（次節以降で学ぶ）に関する同様の結果の物まねである．たとえば，正則言語に対する反復補題は，Y. Bar-Hillel, M. Perles と E. Shamir

[1] による文脈自由言語に対する類似の結果を単純化したものである．同じ論文には，直接的にではないが，本節で取り上げた他のいくつかの閉包性も与えられている．ただし，逆準同型写像のもとでの閉包性は [2] による．

問 4.2.2 で紹介した商の演算は [3] から取ったものである．なお，その文献では，一つの文字 a の代わりに任意の正則表現を用いる，もっと一般的な演算が議論されている．問 4.2.8 では正則言語に属す文字列の前半分から成る言語に注目したが，問 4.2.8 以降に取り上げたそのような「部分除去」型の一連の演算に関する考察は [8] から始まった．J. I. Seiferas と R. McNaughton [9] は，除去型の演算が正則言語を保存するのはどんな場合であるかという一般的な問題を研究した．

正則言語に関する空言語問題，有限性問題，所属問題などの決定問題を解くアルゴリズムの原形は [7] による．DFA の状態数を最小化するアルゴリズムは [7] にあるが，[5] にも記述されている．状態数が最小の DFA を見つけるアルゴリズムでもっとも効率のよいものは [4] に記されている．

[1] Y. Bar-Hillel, M. Perles, and E. Shamir, “On formal properties of simple phrase-structure grammars,” *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14** (1961), pp.143-172.

[2] S. Ginsburg and G. Rose, “Operations which preserve definability in languages,” *J. ACM* **10**:2 (1963), pp.175-195.

[3] S. Ginsburg and E. H. Spanier, “Quotients of context-free languages,” *J. ACM* **10**:4 (1963), pp.487-492.

[4] J. E. Hopcroft, “An $n \log n$ algorithm for minimizing the states in a finite automaton,” in Z. Kohavi (ed.) *The Theory of Machines and Computations,* Academic Press, New York, 1971, pp.189-196.

[5] D. A. Huffman, “The synthesis of sequential switching circuits,” *J. Franklin Inst.* **257**:3-4 (1954), pp.161-190 and 275-303.

[6] S. C. Kleene, “Representation of events in nerve nets and finite automata,” in C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp.3-42.

[7] E. F. Moore, “Gedanken experiments on sequential machines,” in C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp.129-153.

[8] R. E. Stearns and J. Hartmanis, “Regularity-preserving modifications of regular expressions,” *Information and Control* **6**:1 (1963), pp.55-69.

[9] J. I. Seiferas and R. McNaughton, “Regularity-preserving modifications,” *Theoretical Computer Science* **2**:2 (1976), pp.147-154.

5 文脈自由文法と言語

ここで正則言語から,「文脈自由言語」と呼ばれる，より広い言語のクラスに目を転じよう．この言語には「文脈自由文法」と呼ばれる自然な再帰的記法がある．文脈自由文法は 1960 年代以降，コンパイラ技術で中心的な役割を果たしてきた；この文法のお陰で，パーサー（プログラムの構造を見つけだす機能）の構築は，時間のかかるその場しのぎの作業から，半日ですむ定型的な仕事に変わった．その後，文脈自由文法は，ウェブによる情報交換のための XML (eXtensible Markup Language，拡張可能マークアップ言語）の世界で使われているいわゆる文書型定義（Document Type Definition）を通して，文書の体裁を記述するのに使われている．

この章では，まず文脈自由文法の記法を導入し，それがどのように言語を定義するかを説明する．それから文法がその言語の文字列に定める構造を表す図，すなわち「構文木」を論じる．構文木はプログラミング言語のためのパーサーの出力であって，プログラムの構造がよく捉えられているものである．

「プッシュダウン・オートマトン」と呼ばれる,（有限）オートマトンによく似た記法がある．これも文脈自由言語を，そしてそれだけを記述できる；プッシュダウン・オートマトンは第 6 章で導入される．有限オートマトンほど重要ではないがプッシュダウン・オートマトンは，言語を定義する機構として文脈自由文法と等価であるため，第 7 章で論じる文脈自由言語の閉包性や決定問題を考察するのに非常に役に立つ．

5.1 文脈自由文法

まず文脈自由文法の概念を直観的に導入することから始めよう．この文法の重要な能力をいくつか観察してから，形式的な定義を与える．文法を形式的にどのように定義するかを示し，その文法が定める言語の文字列を決定する「導

出」過程を導入する．

5.1.1 直観的な例

回文について考えてみよう．**回文**とは，`otto` や `madamimadam`（Madam, I'm Adam：エデンの園で，イヴが初めて聞いた言葉とか）のように，前から読んでも後ろから読んでも変わらない文字列のことである．いい方を変えれば，w が回文である必要十分条件は，$w = w^R$ が成り立つことである．話を簡単にするために，アルファベット $\{0, 1\}$ 上の回文だけを考えることにしよう．この言語（回文の集合）は 0110 や 11011，ε を含むが，011 や 0101 は含まない．

0 と 1 から成る回文の言語 L_{pal} は，正則言語ではない．これを確かめるには，「反復補題」を使う．L_{pal} が正則言語であると仮定する：関係する定数を n として，回文 $w = 0^n10^n$ を考える．もし L_{pal} が正則なら，w は $w = xyz$ という形に分解でき，y は左側の 0 を少なくとも一つ含む（しかも 1 や右側の 0 を含まない）．したがって，xz は L_{pal} が正則ならやはり L_{pal} に属しているはずであるが，1 より左側の 0 は 1 より右側の 0 よりも少ない．したがって xz は回文ではあり得ない．これは「L_{pal} が正則言語である」という仮定に矛盾する．

0 と 1 の列がいつ L_{pal} に入るかについて，自然で再帰的な定義がある．いくつかの明らかに L_{pal} の中にある列を基礎として，それから「ある列が回文であれば，最初と最後の文字は同じである」ことを利用する．さらに，最初と最後の文字を除いた残りもやはり回文になる．すなわち：

基礎：ε，0，および 1 は回文である．

再帰：もし w が回文なら，$0w0$ と $1w1$ も回文である．0 と 1 で作れる回文は，これらの規則（基礎と再帰）で構成できるものに限る．

文脈自由文法はこのような再帰的定義を形式的に記述する記法である．文法はある文字列のクラス，すなわち言語を代表するいくつかの変数を含む．この例では，変数は「回文の集合」つまり「言語 L_{pal} を構成する文字列のクラス」を表す P 一つだけでよい．各クラスの中の文字列がどのように構成されるかを示す規則もある．その規則は変数とアルファベットの中の記号を使って記述される．

例 5.1　回文を定義する規則は，文脈自由文法の記法で表すと，図 5.1 のようになる．この規則の意味は，5.1.2 項で説明する．

最初の三つの規則は「基礎」である．それらは L_{pal} が列 ε，0，および 1 を含んでいることを示している．これら三つの規則の右辺（矢印のあとに続く部分）には変数がない——それこそ再帰的定義の基礎になる理由である．

最後の二つの規則は定義の再帰的部分である．たとえば規則 4 は，「もし w がクラス P の中にあれば，$0w0$ もクラス P の中にある」ことをいっている．規則 5 も同じように，$1w1$ が P の中にあることを主張している．▨

1. $P \to \varepsilon$
2. $P \to 0$
3. $P \to 1$
4. $P \to 0P0$
5. $P \to 1P1$

図 5.1　回文に対する文脈自由文法

5.1.2 文脈自由文法の定義

言語の文法的記述には四つの重要な構成要素がある．

(1) 定義される言語の文字列を構成する記号の有限集合．さっきの回文の例では，これは $\{0, 1\}$ であった．我々はこのアルファベットを**終端記号**の集合と呼ぶ．

(2) **変数**の有限集合．なお変数は**非終端記号**あるいは**文法概念**とも呼ばれる．各変数はある言語，すなわち文字列の集合を表している．上の例では，変数は P だけで，これがアルファベット $\{0, 1\}$ 上の回文のクラスを表していた．

(3) 定義しようとしている言語を表す，特別の記号；それは**出発記号**と呼ばれる．それ以外の変数は出発記号が表す言語を定義するのを助ける，補助的な文字列のクラスを表している．上の例ではただ一つの変数 P が出発記号である．

(4) **生成規則**（単に**規則**とも呼ばれる）の有限集合．生成規則は言語の再帰的定義を表現しているので，次の部分からできている．

（ア）その生成規則によって（部分的に）定義される変数．この変数はよく，生成規則の**頭部**と呼ばれる．

（イ）生成記号 $\rightarrow$

（ウ）0 個以上の終端記号と変数から成る列．この列は，生成規則の**本体**と呼ばれ，頭部が表している言語の中の列を構成する一つの方法を表している．生成規則の適用によって，終端記号は変わらないが，本体の中の各変数は（他の規則によって書き換えられ，最終的には）その変数が表す言語の文字列に置き換えられる．

生成規則の例は図 5.1 にある．

ここに示した四つの構成要素が，**文脈自由文法**，あるいは **CFG**（あるいは単に**文法**）を定める．これら四つの構成要素，正確にいえば変数の集合 V，終端記号の集合 T，生成規則の集合 P，それに出発記号 S を指定すれば一つの CFG G が定まるので，G は次のように表現される：$G = (V, T, P, S)$

例 5.2 回文を表す文法 G_{pal} は，次のように表現できる：

$$G_{pal} = (\{P\}, \{0, 1\}, A, P)$$

ここで A は図 5.1 に示された五つの生成規則の集合を表す．▨

例 5.3 もっと複雑な文脈自由文法——典型的なプログラミング言語の（単純化された）式を表現する CFG を取り上げてみよう．まず，演算は加算と乗算を表す $+$ と $*$ だけに限る．非演算項として識別子（identifier）を許すが，典型的な識別子（英字に始まる長さ 1 以上の英数字の列）をすべて許すのでなく，文字 a，b と数字 0，1 に限定する．どの識別子も a または b に始まり，そのあとに $\{a, b, 0, 1\}^*$ の任意の列をつけ加えた列である．

今度の文法には二つの変数が必要である．一つは E で，式を表す．これは定義しようとしている式の集合（言語）を表す出発記号である．もう一つは I で，識別子を表す．その言語は実は正則である．この言語の正則表現は：

$$(\mathbf{a}+\mathbf{b})(\mathbf{a}+\mathbf{b}+\mathbf{0}+\mathbf{1})*$$

しかし，文法の中では正則表現は直接的には使わない．その代わり，この正則表現と実質的に同じことを表す生成規則の集合を使う．

式を表すこの文法は，形式的には $G = (\{E, I\}, T, P, E)$ で定義される：ここで T は記号の集合 $\{+, *, (,), a, b, 0, 1\}$ で，P は図 5.2 に示された生成規

$$
\begin{array}{ll}
(1) & E \rightarrow I \\
(2) & E \rightarrow E + E \\
(3) & E \rightarrow E * E \\
(4) & E \rightarrow (E) \\
(5) & I \rightarrow a \\
(6) & I \rightarrow b \\
(7) & I \rightarrow Ia \\
(8) & I \rightarrow Ib \\
(9) & I \rightarrow I0 \\
(10) & I \rightarrow I1
\end{array}
$$

図 5.2　単純な式を表す文脈自由文法

則の集合である．生成規則は次のように解釈される．

規則 1 は式の基礎である．これは「式は単一の識別子であり得る」ことを示している．規則 (2) から (4) までは式の再帰的な構成法を述べている．規則 (2) によれば，二つの式を加算記号 + でつないだものもやはり式である．規則 (3) は同じことを乗算記号 $*$ について述べている．規則 (4) は，任意の式の両側を対応する括弧で囲んだものもやはり式である，といっている．

規則 (5) から (10) までは識別子を記述している．基礎は規則 (5) と (6) である；これらは a と b が識別子である，といっている．残りの四つは再帰的な場合を扱っている：任意の識別子に対して，そのあとに a, b, 0, 1 をつけ加えたものも識別子である．▨

生成規則の簡潔な表現

生成規則は頭部の変数ごとにまとめて考えると都合がよい．頭部が変数 A である生成規則を「A のための生成規則」とか「A–生成規則」などと呼ぶことが多い．ある文法の生成規則を各変数ごとに 1 回だけ書き，その変数に対するすべての本体をタテ棒で区切って列挙することもある．すなわち，生成規則 $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, $\cdots$, $A \rightarrow \alpha_n$ は $A \rightarrow \alpha_1 \,|\, \alpha_2 \,|\, \cdots \,|\, \alpha_n$ と書くことができる．たとえば図 5.1 に示した回文のための文法は，$P \rightarrow \varepsilon \,|\, 0 \,|\, 1 \,|\, 0P0 \,|\, 1P1$ と書ける．

5.1.3 文法による導出

CFG の生成規則を適用するのは，ある文字列がある変数の言語に属していることを判定するためである．この判定には二つの方法がある．一般的な方法は，規則を「本体から頭部へ」と使うことである．すなわち，本体の中にある各変数の言語に属しているとわかっている文字列を選び，それらを終端記号と合わせて正しい順序につないで，その結果得られる列が「頭部の変数の言語に属する」と推論するのである．この手続きは**再帰的推論**と呼ばれる．

文法の言語を定義するもう一つの方法は，生成規則を「頭部から本体へ」と使用する．出発記号をその生成規則（出発記号を頭部とする生成規則）のどれかで展開する（本体に置き換える）．そしてその結果得られる列を，その中の変数のどれかをその生成規則の本体と置き換えることによって，さらに展開する——これを終端記号だけから成る文字列に到達するまで続ける．その文法の言語とは，そのようにして得られるすべての終端記号列の集合である．文法のこの使い方（展開の繰り返し）を**導出**という．

ここでは最初の方法——再帰的推論から始めよう．しかし，文法を導出の道具として考える方が自然なので，あとで導出を記述するための記法を開発しておく．

例 5.4 図 5.2 の文法を使ってできる推論について少し考えてみよう．図 5.3 はその推論の要約である．たとえば，第 (i) 行は，a が I に対する言語の中にあることを，推論規則 (5) から推論できることをいっている．(ii) から (iv) までの行は，$b00$ が識別子であることが，生成規則 (6) を（b を得るために）1 回使い，生成規則 (9) を（0 をつけ加えるために）2 回使うことによって推論できることをいっている．

(v) 行と (vi) 行は，生成規則 (1) を使って，どんな識別子も式であるから，(i)，(iv) によって識別子であるとすでに推論ずみの文字列 a および $b00$ が「変数 E の言語の中にある」ことを推論している．(vii) 行は生成規則 (2) によってこれらの識別子（すなわち式）の和が式であることを推論している；(viii) 行は生成規則 (4) によって，式を括弧で囲んだものも式であることが推論される；(ix) 行では生成規則 (3) を，識別子 a（第 (vi) 行で式であると推論ずみ）と第 (viii) 行で推論した式との積を求めるのに使っている．▨

	導かれる文字列	何の言語か	使われた生成規則	使われた文字列
(i)	a	I	5	—
(ii)	b	I	6	—
(iii)	$b0$	I	9	(ii)
(iv)	$b00$	I	9	(iii)
(v)	a	E	1	(i)
(vi)	$b00$	E	1	(iv)
(vii)	$a+b00$	E	2	(v), (vi)
(viii)	$(a+b00)$	E	4	(vii)
(ix)	$a*(a+b00)$	E	3	(v), (viii)

図 5.3　図 5.2 の文法による文字列の推論

生成規則を頭部から本体へと適用して文字列を導く過程を記述するには，新しい関係記号 $\Rightarrow$ を定義するとよい．$G=(V, T, P, S)$ を CFG とする．また $\alpha A\beta$ を，変数 A を含む終端記号と変数の列とする．いいかえれば，α と β は $(V\cup T)^*$ の中の列であり，A は V の中にある．$A\rightarrow\gamma$ が G の生成規則であるとき,「G において，$\alpha A\beta$ から $\alpha\gamma\beta$ が導出できる」といって，次のように書く．$\alpha A\beta \underset{G}{\Rightarrow} \alpha\gamma\beta$. G がわかっているときは，単に $\alpha A\beta\Rightarrow\alpha\gamma\beta$ と書く．1 回の導出ステップでは，列の任意の場所の一つの変数が，その変数の生成規則の本体に置き換えられる．

この関係 $\Rightarrow$ は，0，1，あるいはそれ以上のステップの導出に拡張できる——有限オートマトンの推移関数 δ を $\hat{\delta}$ に拡張したのと同じようなことである．導出の場合,「0 回以上のステップ」を * で表し，次のように定義する：

基礎：終端記号と変数の任意の列 α について，$\alpha \underset{G}{\overset{*}{\Rightarrow}} \alpha$ すなわち，どの列も自分自身を導出する．

再帰：もし $\alpha \underset{G}{\overset{*}{\Rightarrow}} \beta$ で $\beta \underset{G}{\Rightarrow} \gamma$ ならば，$\alpha \underset{G}{\overset{*}{\Rightarrow}} \gamma$，すなわち，$\alpha$ から 0 回以上の導出で β が得られ，β が 1 回の導出で γ になるのなら，α から β が導出できる．詳しくいうと $\alpha \underset{G}{\overset{*}{\Rightarrow}} \beta$ とは，ある $n\geq 1$ に対して一連の文字列 $\gamma_1, \gamma_2, \cdots, \gamma_n$ があって，次の条件が成り立つことである．

(1)　$\alpha=\gamma_1$,

(2)　$\beta=\gamma_n$,

(3)　$i=1, 2, \cdots, n-1$ に対して $\gamma_i\Rightarrow\gamma_{i+1}$

文法 G がわかっていれば，$\underset{G}{\overset{*}{\Rightarrow}}$ の代わりに $\overset{*}{\Rightarrow}$ を使う．

例 5.5 $a*(a+b00)$ が変数 E の言語の中にあることの推論は，E に始まる導出に反映される．そのような導出の一つの例を示す：

$$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow$$

$$a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow$$

$$a*(a+I) \Rightarrow a*(a+I0) \Rightarrow a*(a+I00) \Rightarrow a*(a+b00)$$

最初のステップで，E が（図 5.2 の）生成規則（3）の本体に置き換えられる．次のステップで，生成規則（1）を使って（左側の）E を I に書き換え，以下同様に続く．ここでは「いつも列の中の最も左にある変数を書き換える」という組織的方針をとっている．しかし，どのステップにおいても，どの変数をどの規則で書き換えるかは自由に選んでよい．たとえば第 2 のステップで，右側の E を生成規則（4）によって (E) に置き換えることもできた．なお書き換えの仕方を変えると，同じ終端記号列が導けるとは限らない．たとえば最初のステップで生成規則（2）を使えば，$E \Rightarrow E+E$ となるが，これら二つの E をどう書き換えても $a*(a+b00)$ は出てこない．

この導出を，記号 $\overset{*}{\Rightarrow}$ によって圧縮できる．まず基礎によって $E \overset{*}{\Rightarrow} E$ である．再帰のステップを反復適用すれば，$E \overset{*}{\Rightarrow} E*E$，$E \overset{*}{\Rightarrow} I*E$，等々が得られ，最終的に $E \overset{*}{\Rightarrow} a*(a+b00)$ が導かれる．

二つの見方――再帰的推論と導出は等価である．すなわち，終端記号の列 w がある変数 A の言語の中にある必要十分条件は，$A \overset{*}{\Rightarrow} w$ が成り立つことである．しかし，この事実の証明には少し準備が要るので，5.2 節で扱うことにする．▨

5.1.4 最左導出と最右導出

列を生成する選択肢を限定するために，各ステップで最も左の変数を，その変数の一つの生成規則の本体で置き換えることにすると便利なことがよくある．そのような導出は**最左導出**と呼ばれ，1 回か多数回かにより，それぞれ関係記号 $\underset{\text{左}}{\Rightarrow}$ あるいは $\underset{\text{左}}{\overset{*}{\Rightarrow}}$ で表される．もし文法 G を明示する必要があれば，これらの矢印の下に名前 G を明記する．

同様に，書き換えを「最も右の変数」をその生成規則の本体で置き換えることにしてもよい．その場合，導出は**最右導出**と呼ばれ，1 回あるいは多数回の最右導出はそれぞれ $\underset{\text{右}}{\Rightarrow}$ あるいは $\overset{*}{\underset{\text{右}}{\Rightarrow}}$ で表される．どの文法を使っているかを明らかにしたいときはやはり，文法の名前を矢印の下に書く．

例 5.6 例 5.5 の導出は，実は最左導出であった．だから同じ導出を，次のように書いてもよい．

$$E \underset{\text{左}}{\Rightarrow} E * E \underset{\text{左}}{\Rightarrow} I * E \underset{\text{左}}{\Rightarrow} a * E \underset{\text{左}}{\Rightarrow}$$
$$a * (E) \underset{\text{左}}{\Rightarrow} a * (E + E) \underset{\text{左}}{\Rightarrow} a * (I + E) \underset{\text{左}}{\Rightarrow} a * (a + E) \underset{\text{左}}{\Rightarrow}$$
$$a * (a + I) \underset{\text{左}}{\Rightarrow} a * (a + I0) \underset{\text{左}}{\Rightarrow} a * (a + I00) \underset{\text{左}}{\Rightarrow} a * (a + b00)$$

この最左導出を要約して，$E \overset{*}{\underset{\text{左}}{\Rightarrow}} a * (a + b00)$ と書くこともできる．また途中のいくつかのステップをまとめて，たとえば $E * E \overset{*}{\underset{\text{左}}{\Rightarrow}} a * (E)$ と書いてもよい．

CFG 導出の記法

CFG を扱っているとき，記号の役割をわかりやすくするための，いくつか便利な習慣がある．次に我々が利用する習慣を挙げておく：

(1) アルファベットの最初の方の小文字 a，b などは終端記号である．数字や +，括弧などの記号も終端記号である．

(2) アルファベットの最初の方の大文字 A，B などは変数である．

(3) アルファベットの後ろの方の小文字 w，z などは，終端記号の列を表す．この習慣は，終端記号がオートマトンの入力列に対応していることを思い出させてくれる．

(4) アルファベットの後ろの方の大文字 X，Y などは，終端記号または変数を表す．

(5) 小文字のギリシャ文字 α，β などは，終端記号と変数の一方あるいは両方を含む列を表す．

変数だけから成る列には，あまり重要な意味がないので，特別の記法はない．しかし α などのギリシャ文字で表された列が，たまたま変数だけから成ることはあり得る．

各変数について，順序は異なるが，まったく同じ書き換えを行う最右導出がある．次の導出がそうである：

$$E \underset{右}{\Rightarrow} E*E \underset{右}{\Rightarrow} E*(E) \underset{右}{\Rightarrow} E*(E+E) \underset{右}{\Rightarrow}$$
$$E*(E+I) \underset{右}{\Rightarrow} E*(E+I0) \underset{右}{\Rightarrow} E*(E+I00) \underset{右}{\Rightarrow} E*(E+b00) \underset{右}{\Rightarrow}$$
$$E*(I+b00) \underset{右}{\Rightarrow} E*(a+b00) \underset{右}{\Rightarrow} I*(a+b00) \underset{右}{\Rightarrow} a*(a+b00)$$

この導出から，$E \underset{右}{\overset{*}{\Rightarrow}} a*(a+b00)$ と結論できる．▨

どんな導出にも，それと等価な最左導出と最右導出がある．すなわち，w が終端記号列で A が変数であるとき，$A \overset{*}{\Rightarrow} w$ であるための必要十分条件は $A \underset{左}{\overset{*}{\Rightarrow}} w$ であり，また $A \overset{*}{\Rightarrow} w$ である必要十分条件は $A \underset{右}{\overset{*}{\Rightarrow}} w$ である．これらの主張も 5.2 節で証明する．

5.1.5 ある文法の言語

もし $G = (V, T, P, S)$ が CFG ならば，文法 G の言語 $L(G)$ とは，出発記号から導出できる終端記号列の集合のことである．すなわち，

$$L(G) = \{w \in T^* \mid S \underset{G}{\overset{*}{\Rightarrow}} w\}$$

言語 L がある文脈自由文法の言語であるとき，L は**文脈自由言語**，あるいは **CFL** と呼ばれる．たとえばすでに見たように，図 5.1 の文法は $\{0, 1\}$ 上の回文の言語を定義するので，回文の集合は文脈自由言語である．この事実は定理として，次のように証明できる．

定理 5.7 例 5.1 の文法 G_{pal} の言語 $L(G_{pal})$ は，$\{0, 1\}$ 上の回文の集合である．

証明 $\{0, 1\}*$ の中の列 w が $L(G_{pal})$ に属する必要十分条件は，w が回文であること，すなわち $w = w^R$ であることを示せばよい．

（十分性）w が回文であると仮定する．我々は長さ $|w|$ についての帰納法によって，w が $L(G_{pal})$ に属していることを示す．

基礎：長さ 0 と 1 の列を基礎とする．もし $|w| = 0$ または $|w| = 1$ ならば，w は ε, 0, 1 のどれかである．生成規則 $P \to \varepsilon$, $P \to 0$, $P \to 1$ があるので，これらはどれも $L(G_{pal})$ に属している．

帰納：$|w| \geq 2$ と仮定する．$w = w^R$ であるから，w の最初と最後の記号は同じである．すなわち，$w = 0x0$ または $w = 1x1$ である．さらに，$x = x^R$ から，x も回文で

なければならない．このように $|w| \geq 2$ という事実から，w の両端に二つの 0，あるいは二つの 1 があることが導かれる．

もし $w = 0x0$（でしかも x が回文）であれば，帰納法の仮定から $P \overset{*}{\Rightarrow} x$ となる．すると w も P から生成できる——実際 $P \overset{*}{\Rightarrow} x \Rightarrow 0x0 = w$ である．もし $w = 1x1$ であっても同様で，ただ最初のステップで生成規則 $P \to 1P1$ を使えばよい．どちらの場合も w は $L(G_{pal})$ の中にあるので，証明が終わる．

（必要性）今度は w が $L(G_{pal})$ の中にある，すなわち $P \overset{*}{\Rightarrow} w$ と仮定する．この w が回文であることを示さなければならない．これは，P から w を導く導出のステップ数についての帰納法で証明できる．

基礎：導出が 1 ステップであれば，本体に P を含まない三つの生成規則のどれかを使うしかない．すなわち導出は $P \Rightarrow \varepsilon$, $P \Rightarrow 0$ または $P \Rightarrow 1$ である．$\varepsilon, 0, 1$ はどれも回文であるから，基礎は証明できた．

帰納：導出が $n+1$ ステップ（$n \geq 1$）で，n ステップ以下の導出については「必要性」が成り立つと仮定する．すなわち，もし n ステップで $P \overset{*}{\Rightarrow} x$ となるならば，x は回文である．

さて，w の $(n+1)$ ステップでの導出を考えると，$n+1$ ステップとは 2 ステップ以上なので，最初は P を含む生成規則，つまり $P \to 0P0$ か $P \to 1P1$ のどちらかでなければならない．したがって導出は

$$P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$$

あるいは

$$P \Rightarrow 1P1 \overset{*}{\Rightarrow} 1x1 = w$$

のどちらかの形をしている．どちらの場合も，$P \overset{*}{\Rightarrow} x$ は n ステップである．

帰納法の仮定から，x は回文であることがわかる；つまり $x = x^R$ である．しかしもしそうだとすれば，$0x0$ と $1x1$ もやはり回文である：たとえば $(0x0)^R = 0x^R0 = 0x0$．したがって w は回文であり，必要性の証明が終わる．▨

5.1.6 文　形　式

開始記号からの導出は，ある特別な役割を持った記号列を作り出す．それらは「文形式」と呼ばれる．すなわち，$G = (V, T, P, S)$ が CFG であれば，$(V \cup T)^*$ に属し，$S \overset{*}{\Rightarrow} \alpha$ を満たすどんな列 α も**文形式**である．もし $S \underset{左}{\overset{*}{\Rightarrow}} \alpha$ ならば，α は左文形式と呼ばれる．もし $S \underset{右}{\overset{*}{\Rightarrow}} \alpha$ ならば，α は右文形式と呼ばれる．言語 $L(G)$ とは，T^* に属する，つまり T の要素だけから成る文形式の集合である．

例 5.8 図 5.2 に示した，数式のための文法を考えよう．たとえば

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

という導出があるので，$E * (I + E)$ は文形式である．しかし最後のステップで中央の E が書き換えられているので，これは最左導出でも最右導出でもない．

左文形式の例としては，最左導出

$$E \underset{左}{\Rightarrow} E * E \underset{左}{\Rightarrow} I * E \underset{左}{\Rightarrow} a * E$$

から，$a * E$ がそうである．また最右導出

$$E \underset{右}{\Rightarrow} E * E \underset{右}{\Rightarrow} E(E) \underset{右}{\Rightarrow} E * (E + E)$$

から，$E * (E + E)$ は右文形式であることがわかる． ▨

文法についての証明の形

定理 5.7 の証明は，ある文法が直観的に定義された特定の言語を定めることを示す，典型的な例である．まず各変数から導かれる記号列の性質に注目し，それを帰納法の仮定として定式化する．この例では，変数は P だけなので，P から導かれる列が回文であることを示せばよい．

「十分」であることの証明：一つの変数 A から導かれる列についての直観的な記述を w が満たしているなら，$A \overset{*}{\Rightarrow} w$ であることを証明する．今の場合，P は出発記号なので，$A \overset{*}{\Rightarrow} w$ とは w がその文法の言語に属することに他ならない．ふつう，この部分は w の長さについての帰納法で証明される．もし k 個の変数があれば，証明すべき文は k 個の部分に分かれ，相互的帰納法で証明することになる．

「必要」であることの証明：もし $A \overset{*}{\Rightarrow} w$ ならば，w は変数 A から導かれる列についての直観的な記述を満たしていることを証明しなければならない．今考えている例では，出発記号 P だけを考えればよいので，$P \overset{*}{\Rightarrow} w$ とは「w が言語 G_{pal} に属する」と言い換えられる．この部分の証明には，導出のステップ数についての帰納法を使うことが多い．二つ以上の変数が現れるような生成規則を持つ文法では，n ステップの導出を「それぞれの変数からの導出」に分ける必要がある．それらの導出は n ステップより短いので，1.4.2 項で述べたように，n より小さいステップ数に対して仮定が正しいとする帰納法を用いるとよい．

5.1.7　5.1 節の演習問題

♣ **問 5.1.1**　次の言語に対する文脈自由文法を作れ．

*****a)　$\{0^n1^n \mid n \geq 1\}$，すなわち「1 個以上の 0 のあとに同じ個数の 1 を続けた列」の集合．

***!**b)　$\{a^ib^jc^k \mid i \neq j$ または $j \neq k\}$，すなわち「a の列，b の列，c の列がその順に並び，a と b の数が異なるか，または b と c の数が異なるような列」の集合．

!c)　a と b の列で，ww という形でないもの，すなわち「同じ列の繰り返しでないような列」の集合．

!!d)　0 の個数が 1 の個数の 2 倍であるような列の集合．

♣ **問 5.1.2**　次の文法は正則表現 $\mathbf{0^*1(0+1)^*}$ と同じ言語を生成する：

$$S \to A1B$$

$$A \to 0A \mid \varepsilon$$

$$B \to 0B \mid 1B \mid \varepsilon$$

次の列の最左導出と最右導出を示せ．

*****a)　00101

b)　1001

c)　00011

♣ **! 問 5.1.3**　任意の正則言語は文脈自由言語であることを示せ．
ヒント：正規表現の中の演算子の数についての帰納法によって，CFG を組み立てる．

♣ **! 問 5.1.4**　CFG は，どの生成規則の本体も高々一つの変数を持ち，しかもその変数は右端にあるとき，**右線形**であるという．だから**右線形文法**の規則は，$A \to w$ かまたは $A \to wB$ の形をしている——ここで A，B は変数，w は 0 個以上の終端記号の列である．

a)　右線形文法はどれも正則言語を生成することを示せ．
ヒント：最左導出をシミュレートする ε-NFA を構成する——左文形式の中のただ一つの変数を「状態」にすればよい．

b)　任意の正則言語は，ある右線形文法で生成できることを示せ．
ヒント：DFA から出発し，その状態を変数に置き換えればよい．

♣ **問 5.1.5**　$T = \{0, 1, (,), +, *, \emptyset, e\}$ とおく．T はアルファベット $\{0, 1\}$ 上の正則表現に使われる記号の集合と考えることができる．ただ違うのは，混同を避けるために ε の代わりに e を使うことだけである．課題は T を終端記号の集合とする CFG で，アルファベット $\{0, 1\}$ 上の正則表現の集合を生成するものを作る

ことである．

♣ **問 5.1.6**　前に関係 $\stackrel{*}{\Rightarrow}$ を，基礎「$\alpha \Rightarrow \alpha$」と帰納法「$\alpha \stackrel{*}{\Rightarrow} \beta$ で $\beta \Rightarrow \gamma$ ならば $\alpha \stackrel{*}{\Rightarrow} \gamma$」によって定義した．しかし「$\Rightarrow$ とは 0 ステップ以上の $\Rightarrow$ を表す」と定義する他の方法がある：次のことを証明せよ．

a)　$\alpha \stackrel{*}{\Rightarrow} \beta$ が成り立つ必要十分条件は，1 個以上の記号列

$$\gamma_1, \gamma_2, \cdots, \gamma_n$$

があって，$\alpha = \gamma_1, \beta = \gamma_n$ でしかも，$j = 1, 2, \cdots, n-1$ について

$$\gamma_j \Rightarrow \gamma_{j+1}$$

を満たすことである．

b)　もし $\alpha \stackrel{*}{\Rightarrow} \beta$ で $\beta \stackrel{*}{\Rightarrow} \gamma$ ならば，$\alpha \stackrel{*}{\Rightarrow} \gamma$

ヒント：導出 $\beta \stackrel{*}{\Rightarrow} \gamma$ のステップ数についての帰納法を使え．

♣ **! 問 5.1.7**　生成規則

$$S \rightarrow aS \,|\, Sb \,|\, a \,|\, b$$

で定義される CFG を考える．

a)　$L(G)$ の中のどの列も，部分列として ba を含まないことを証明せよ．

b)　$L(G)$ の内容を直観的に記述せよ．その答を，(a) によって正当化せよ．

♣ **! 問 5.1.8**　生成規則

$$S \rightarrow aSbS \,|\, bSaS \,|\, \varepsilon$$

で定義される CFG を考える．$L(G)$ が，同じ個数の a と b を含む列の集合であることを証明せよ．

5.2 構　文　木

導出を表現する非常に便利な「木表現」がある．その木は，終端記号列の中の記号がどのようにグループ分けされ，文法の変数に結びつけられるかを示してくれる．しかしおそらくさらに重要なのは，この木がコンパイラの中では「構文木」と呼ばれて，ソース・プログラム（利用者が書いた原プログラム）を表現するためのデータ構造として使われていることである．コンパイラはソース・プログラムの木構造のおかげで，ソース・プログラムから実行可能な命令語への翻訳の過程を，自然な帰納的関数で表現することができ，翻訳の仕事がずっと容易になる．

この節では構文木の概念を導入して，構文木の存在が導出の存在と（構文の）再帰的推論に密接に関連していることを示す．あとで文法や言語の「あいまい

さ」を論じるが，これは構文木の重要な応用例である．ある文法は同じ終端記号列が二つ以上の構文木を持つことを許す．そのような状況があるプログラムについて起こると，コンパイラはそのプログラムの構造を決められず，そのプログラムからどのような実行可能な機械語命令群を作ればよいのか定かでなくなるので，あいまいさはプログラミング言語では許されない．

5.2.1 構文木の構成

文法 $G = (V, T, P, S)$ を一つ固定する．G の構文木とは，次の条件を満たす木のことである：

(1) 内側の節点にはどれも，ラベルとして V の中の変数がつけられている．

(2) 葉には変数，終端記号，あるいは空列 ε がラベルとしてつけられている．ただしラベルが ε であるのは，その葉が親の唯一の子である場合に限る．

(3) 内部の節点のラベルが A で，その子のラベルが左から

$$X_1, X_2, \cdots, X_k$$

であるのは，$A \to X_1X_2\cdots X_k$ という生成規則が P の中にある場合に限る．なお X のどれかが ε であるのは，それが唯一の子で，$A \to \varepsilon$ が G の生成規則である場合に限る．

木にかかわる用語

木の概念と，よく使われる用語については読者は慣れていると仮定する．しかし念のため，次に基本用語を列挙しておく．

- 木とは**親子関係**を持つ節点の集合である．どの節点も高々一つの親を持ち，0 個以上の子を持つ：節点の親はその節点より上に描かれ，子は下に描かれる．親と子は線で結ばれる．図 5.4, 5.5, 5.6 は木の例である．
- **根**と呼ばれる，親を持たない節点が一つある．根は木の頂上に描かれる．子のない節点は**葉**と呼ばれる．葉でない節点は**内部節点**と呼ばれる．
- ある節点の子の（子の…）子は，その節点の**子孫**と呼ばれる．ある節点の親の（親の…）親は，その節点の**先祖**と呼ばれる．便宜上，どの節点も自分自身の子孫であり，また先祖である，とする．
- 同じ節点の子は，左から右に一列に順序づけられ，そのように描かれる．節点 N が節点 M の左にあれば，N の子孫は M の子孫より左にある，とみなす．

例 5.9 図 5.4 は，図 5.2 の数式のための文法の一つの構文木を示している．根のラベルは変数 E である．これに適用された生成規則は $E \to E + E$ で，根の 3 人の子のラベルは左から E，$+$，E である．根の左端の子には，規則 $E \to I$ が適用されたので，その節点の子は 1 人だけで，ラベル I がつけられている．▨

例 5.10 図 5.5 は，図 5.1 の回文文法のある構文木を表している．根に適用された生成規則は $P \to 0P0$ で，その中央の子には $P \to 1P1$ が適用された．一番下では生成規則 $P \to \varepsilon$ が使われた．このような使い方のとき，すなわち唯一の子のラベルが ε であるときだけ，ε が構文木に現れる．▨

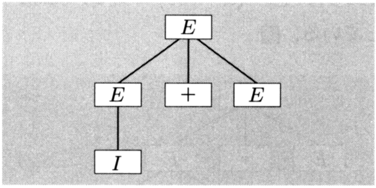


図 5.4 E から $I + E$ への導出を示す構文木

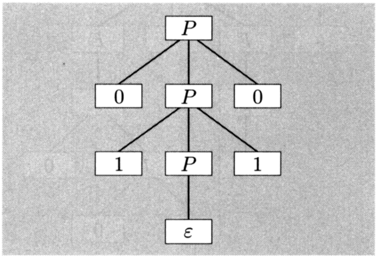


図 5.5 導出 $P \overset{*}{\Rightarrow} 0110$ を示す構文木

5.2.2 構文木の成果

構文木の葉を眺めて，ラベルを左から並べると，文字列が得られる．これはその木の**成果**と呼ばれるが，根の変数から導かれる文形式である．成果が根（の変数）から導出されるという事実は，簡単に証明できる．特に重要なのは，次

のような構文木である：

(1)　成果は終端記号列である．すなわち，葉のラベルがすべて終端記号か ε である．

(2)　根のラベルは出発記号である．

そのような木の成果は，もとになる文法の言語の中の列である．これも簡単に証明できるが，文法の言語は「出発記号を根とし終端記号列を成果とするような構文木の，成果の集合」としても定義できる．

例 5.11　図 5.6 は成果が終端記号列で根が出発記号であるような構文木の例である．これは図 5.2 で導入した，式を定義する文法に基づいている．この木の成果は例 5.5 で導出した列 $a * (a + b00)$ である．実際，この構文木は前の導出をぴったり表現している．▨

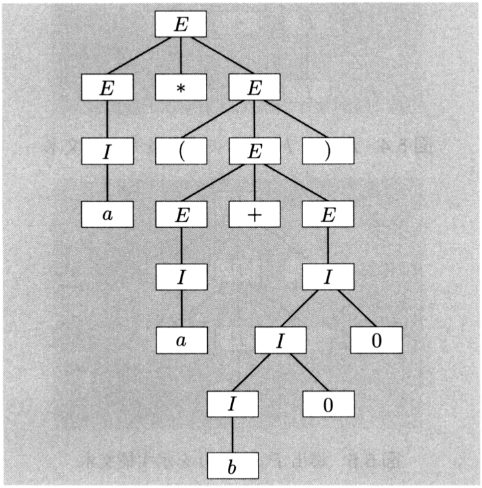


図 5.6　$a * (a + b00)$ が式のための文法の言語に属していることを示す構文木

5.2.3　推論・導出と構文木

これまで文法がどのように働くかを記述するために導入した概念から，列についての対応する事実が導かれる．すなわち，文法 $G = (V, T, P, S)$ に対し

て，次のことは同値である．

(1) 終端列の再帰的な解析（つまり「推論」）によって，変数 A の言語に属する終端記号列 w が決定できる．

(2) $A \overset{*}{\Rightarrow} w$

(3) $A \underset{左}{\overset{*}{\Rightarrow}} w$

(4) $A \underset{右}{\overset{*}{\Rightarrow}} w$

(5) A を根とし w を成果とする構文木がある．

これらは実は，終端記号列だけにかかわる再帰的な推論を除いて，他の条件（導出の存在，最左あるいは最右導出の存在，構文木の存在）はどれも，w が変数を含む列であっても成り立つ．

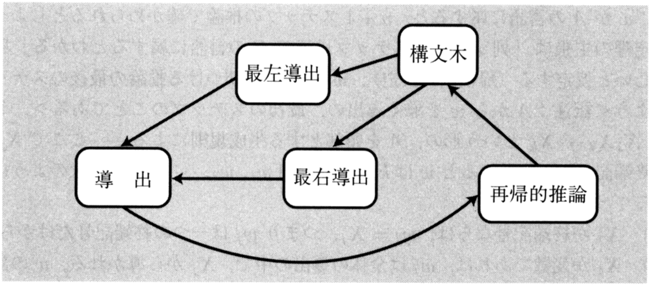


図 5.7 文法についての命題の同値性の証明

これらの同値性を証明したいが，図 5.7 の方針にしたがうことにしよう．すなわち，図の中の矢印ごとに,「w がその根元の条件を満たせば，先の側の条件を満たす」ことを定理として証明する．たとえば定理 5.12 では「w が A の言語に属することが再帰的推論で示せれば，A を根とし w を成果とする構文木が存在する」ことを示す．

二つの矢印は非常に簡単なので，形式的な証明はしない．もし w が A から最左導出で導かれれば，最左導出も一つの導出であるから，w が A から導出できるのは明らかである．同様に，w が最右導出できれば，もちろん導出できる．以下，同値性の証明のよりむずかしいステップに進もう．

5.2.4 推論・導出から木へ

定理 5.12 $G = (V, T, P, S)$ を CFG とする．もし再帰的推論によって w が変数 A の言語に属していることが確かめられるなら，A を根とし w を成果とする構文木が存在する．

証明　w が A の言語に属することを示す導出のステップ数についての帰納法で証明する．

基礎：1 ステップの場合．基礎となる推論だけが実行される．だから $A \to w$ という形の生成規則がなければならない．すると図 5.8 の木が，文法 G の構文木の条件を満たし，明らかに成果は w で根は A である．$w = \varepsilon$ という特殊な場合には，一つの葉にラベル ε がつけられた木ができるが，それも A を根とし w を成果とする，合法的な構文木である．

帰納：w が A の言語に属すると，$n+1$ ステップの推論で確かめられるとしよう．ここで定理の主張は，「列 x が，n ステップ以下で B の言語に属するとわかる」場合には正しいと仮定する（帰納法の仮定）．w を A と結びつける推論の最後のステップを考えよう＜**訳注**：A から w を導く導出の，最初のステップのことである＞．それは $A \to X_1X_2 \cdots X_k$ という形の，A を頭部とする生成規則による——ここで X_j は変数か終端記号である．すると w は k 個の部分列 $w_1, w_2, \cdots, w_k$ に，次のように分解できる：

(1)　X_j が終端記号ならば，$w_j = X_j$，つまり w_j は一つの終端記号だけから成る．

(2)　X_j が変数であれば，w_j は全体の導出の中で，X_j から導かれる，w の部分列である．全体で $n+1$ のステップのうち，最後のステップ $A \to X_1X_2 \cdots X_k$ は関係ないので，w_j と X_j は n ステップ以下の推論で結びつけられる．だから帰納法の仮定を適用することができ，w_j を成果とし X_j を根とする構文木が存在するはずである．

次に A を根とし w を成果とする木を，図 5.9 にしたがって作ってみよう．根のラベルは A で，その子は $X_1, X_2, \cdots, X_k$ であるが，$A \to X_1X_2 \cdots X_k$ は G の生成規則であるから，これは正しい構文木になっている．

各 X_j に対する節点は，w_j を成果とする部分木の根になっている．(1) の場合，つまり X_j· が終端記号であるときは，その部分木は根 X_j だけから成るつまらない木である．(1) の場合には $X_j = w_j$ なので，「成果が w_j である」という条件はもちろん満たされる．

(2) の場合，つまり X_j が変数であるときは，帰納法の仮定から「X_j を根とし，w_j を成果とする構文木が存在する」ことがいえる．その木を X_j に対する節点につないだのが図 5.9 の木である．

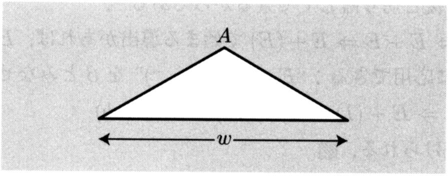


図 5.8 定理 5.12 の「基礎」の場合に構成される木

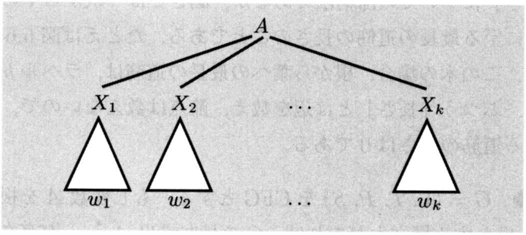


図 5.9 定理 5.12 の証明の，帰納法の部分で使われる木

こうして作られた木の根は A であり，成果は部分木の成果を左から右に順につなげたものである．その結果は文字列 $w_1w_2\cdots w_k$ で，これは w に等しい．▨

5.2.5 木から導出へ

今度は構文木から最左導出を構成するしかたを示そう．最右導出は同じ考え方で構成できるので，ここでは最左導出だけを論じる．どのように導出が構成されるかを理解するには，ある変数からの一つの導出がどのようにして他の導出の中に埋め込まれるかを観察するとよい．次の例が核心部分を示している．

例 5.13 再び図 5.2 で示した式の文法を考える．次の導出ができることはすぐにわかる：

$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab$$

ここから，任意の列 α, β に対して

$$\alpha E\beta \Rightarrow \alpha I\beta \Rightarrow \alpha Ib\beta \Rightarrow \alpha ab\beta$$

であることもわかる．理由は，生成規則の頭部を本体に置き換えることは文脈

α-β の中で，前後と切り離してできるからである[1]．

たとえば $E \Rightarrow E+E \Rightarrow E+(E)$ で始まる導出があれば，E から ab の導出を2番目の E に応用できる："$E+($" を α，"$)$" を β とみなせば

$$E+(E) \Rightarrow E+(I) \Rightarrow E+(Ib) \Rightarrow E+(ab)$$

という導出を続けられる．▨

これでやっと構文木を最左導出に変換する定理の証明に取りかかれる．証明は木の「高さ」についての帰納法であるが，**高さ**とは「根から下の方に，子孫を辿って葉に至る最長の道筋の長さのことである．たとえば図5.6の木の高さは7である．この木の場合，根から葉への最長の道筋は，ラベル b の葉までの道筋である．ふつう「長さ」とは辺を数え，節点は数えないので，ただ一つの節点から成る道筋の長さは0である．

定理 5.14　$G=(V, T, P, S)$ をCFGとする．もし変数 A を根とし終端記号列 w を成果とする構文木があれば，G の最左導出 $A \underset{左}{\overset{*}{\Rightarrow}} w$ が存在する．

証明　木の高さについての帰納法で証明する．

基礎：基礎は，終端記号を生成できる最小の高さ1の場合である．この場合は，木は図5.8のように，根のラベルが A でその子は左から w と読めるとしてよい．これは構文木なので，$A \to w$ が生成規則でなければならない．したがって $A \underset{左}{\Rightarrow} w$ は，A から w への1ステップの最左導出である．

帰納：高さが $n>1$ の木は，図5.9のような形をしている：根のラベルは A であるが，その子のラベルを左から $X_1, X_2, \cdots, X_k$ とする．各 X は終端記号か変数である．

(1)　もし X_j が終端記号なら，w_j を X_j だけから成る列と定める．

(2)　もし X_j が変数なら，それはある終端記号列を成果とする部分木の根のラベルである．そこでその成果を w_j とおく．この場合，部分木の高さは n より小さいので，帰納法の仮定が使える．したがって最左導出 $X_j \underset{左}{\overset{*}{\Rightarrow}} w_j$ が存在する．

もちろん $w=w_1w_2\cdots w_k$ である．

w の最左導出は，次のようにして構成できる．まずステップ $A \underset{左}{\Rightarrow} X_1X_2\cdots X_k$ から始める．次に，各 $j=1, 2, \cdots, k$ の順に

---

1) 実は，「変数を記号列に置き換える作業は文脈と無関係にできる」という性質こそ，用語「文脈自由」が生まれた理由である．「文脈依存」と呼ばれるより強力な文法のクラスがあって，そこでは左あるいは右，あるいは両側にある記号列に依存して，書き換えが許される．文脈依存文法は実用的にはあまり使われていない．

$$A \underset{左}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_j X_{j+1} X_{j+2} \cdots X_k$$

であることを示す．これは別の帰納法，すなわち j についての帰納法で証明できる．まず基礎は $j=0$ の場合，すなわち $A \underset{左}{\Rightarrow} X_1 X_2 \cdots X_k$ で，これはすでに示した．次に

$$A \underset{左}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_{j-1} X_j X_{j+1} \cdots X_k$$

を仮定する．

a) もし X_j が終端記号ならば，何もしなくてよい——$X_j = w_j$ なので，

$$A \underset{左}{\Rightarrow} w_1 w_2 \cdots w_j X_{j+1} X_{j+2} \cdots X_k$$

はすでに得られている．

b) もし X_j が変数ならば，この文脈の中で，X_j から w_j を導く最左導出を続ける．すなわち，その導出が

$$X_j \underset{左}{\Rightarrow} \alpha_1 \underset{左}{\Rightarrow} \alpha_2 \cdots \underset{左}{\Rightarrow} w_j$$

であれば，次のように続けるのである．

$$\begin{aligned}
&w_1 w_2 \cdots w_{j-1} X_j X_{j+1} \cdots X_k \underset{左}{\Rightarrow} \\
&w_1 w_2 \cdots w_{j-1} \alpha_1 X_{j+1} \cdots X_k \underset{左}{\Rightarrow} \\
&w_1 w_2 \cdots w_{j-1} \alpha_2 X_{j+1} \cdots X_k \underset{左}{\Rightarrow} \\
&\cdots \\
&w_1 w_2 \cdots w_{j-1} w_j X_{j+1} \cdots X_k
\end{aligned}$$

このようにして導出 $A \underset{左}{\Rightarrow} w_1 w_2 \cdots w_j X_{j+1} X_{j+2} \cdots X_k$ が得られる．

$j = k$ の場合が，A から w への最左導出である．▨

例 5.15 図 5.6 の木の最左導出を求めてみよう．ここでは根の部分木に対応する導出から木全体に対応する導出を構成する，最終段階だけを示す．すなわち，定理 5.14 で使われた技法の再帰的応用によって，根の最初の子を根とする部分木に対応する導出 $E \underset{左}{\Rightarrow} I \underset{左}{\Rightarrow} a$ や，根の 3 番目の子を根とする部分木に対応する以下の最左導出などはすでに得られているとする：

$$\begin{aligned}
&E \underset{左}{\Rightarrow} (E) \underset{左}{\Rightarrow} (E+E) \underset{左}{\Rightarrow} (I+E) \underset{左}{\Rightarrow} (a+E) \underset{左}{\Rightarrow} \\
&(a+I) \underset{左}{\Rightarrow} (a+I0) \underset{左}{\Rightarrow} (a+I00) \underset{左}{\Rightarrow} (a+b00)
\end{aligned}$$

木全体の最左導出は，根に対応するステップ $A \underset{左}{\Rightarrow} E * E$ に始まり，最初の E

を（そのあとの $*E$ はそのままで）書き換える最左導出を続ける．そこまでの最左導出はだから

$$E \underset{\text{左}}{\Rightarrow} E * E \underset{\text{左}}{\Rightarrow} I * E \underset{\text{左}}{\Rightarrow} a * E$$

となる．終端記号 $*$ は書き換えられないので，これで最初の 2 人の子を根とする部分木の処理がすんだことになる．あとは $E \overset{*}{\underset{\text{左}}{\Rightarrow}} (a + b00)$ にしたがって，それより前後の文脈（前は $a*$，後は空列）はそのままにして，最左導出を完成させればよい．結果は実は例 5.6 に出ているが，以下の通りである：

$$E \underset{\text{左}}{\Rightarrow} E * E \underset{\text{左}}{\Rightarrow} I * E \underset{\text{左}}{\Rightarrow} a * E \underset{\text{左}}{\Rightarrow}$$
$$a * (E) \underset{\text{左}}{\Rightarrow} a * (E + E) \underset{\text{左}}{\Rightarrow} a * (I + E) \underset{\text{左}}{\Rightarrow} a * (a + E) \underset{\text{左}}{\Rightarrow}$$
$$a * (a + I) \underset{\text{左}}{\Rightarrow} a * (a + I0) \underset{\text{左}}{\Rightarrow} a * (a + I00) \underset{\text{左}}{\Rightarrow} a * (a + b00)$$ ▨

同様の定理が最右導出についても成り立つ．木から最右導出を構成する技法は最左導出とほとんど同じで，$A \underset{\text{右}}{\Rightarrow} X_1 X_2 \cdots X_k$ から出発するが，ただ右端の X_k から先に最右導出によって展開し，次に X_{k-1} を展開し，$\cdots$ と X_1 まで続ければよい．そこで証明抜きで定理を述べておく．

定理 5.16 $G = (V, T, P, S)$ を CFG とする．もし変数 A を根とし終端記号列 w を成果とする構文木があれば，G の最右導出 $A \overset{*}{\underset{\text{右}}{\Rightarrow}} w$ が存在する． ▨

5.2.6 導出から再帰的推論へ

次に「ある CFG での導出 $A \overset{*}{\Rightarrow} w$ があれば，w が A の言語に属することを再帰的推論によって確かめられる」ことを証明して，図 5.7 で示した「証明の輪」を完成させよう．定理と証明を述べる前に，導出について一つだいじなことを注意しておこう．

導出 $A \Rightarrow X_1 X_2 \cdots X_k \overset{*}{\Rightarrow} w$ があるとしよう．この w は，$X_j \overset{*}{\Rightarrow} w_j$ であるような w_j に，$w = w_1 w_2 \cdots w_k$ の形に分解できる．なお，もし X_j が終端記号ならば，$w_j = X_j$ で，その間の導出は 0 ステップである．この事実の証明はそうむずかしくない．導出のステップ数についての帰納法で，もし $X_1 X_2 \cdots X_k \overset{*}{\Rightarrow} \alpha$ で $i < j$ ならば，α のうち X_i から導かれるものはどれも，α のうち X_j から導かれるものの左にあることが証明できる．

もし X_i が変数ならば，導出 $A \overset{*}{\Rightarrow} w$ から次の記号とステップを削ることに

よって，導出 $X_i \overset{*}{\Rightarrow} w_i$ が得られる：

a) 導出の過程で現れる文形式の，X_i から導かれる部分より右または左にあるすべての記号.

b) X_i から導かれる部分に関係のないすべてのステップ.

これは次の例から明らかであろう.

例 5.17 図 5.2 の「式の文法」を使った，次の導出を考える.

$$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow I * E + E \Rightarrow I * I + E$$

$$\Rightarrow I * I + I \Rightarrow a * I + I \Rightarrow a * b + I \Rightarrow a * b + a$$

3 番目の文形式 $E * E + E$ と，その中央の変数 E に注目する[2)].

$E * E + E$ から始めて，上の導出の各ステップにしたがいながら，中央の E より左の "$E*$" や，それより右の "$+E$" から導かれる部分を削り落としてみよう．すると各段階での文形式は E, E, I, I, I, b, b になる．このように，第 2 のステップでは中央の E は変わらないので，そのあとのステップで I に変わり，その後の 2 ステップでは変わらず，その次のステップで b に変わる．最後のステップでは中央の E から導かれたものは変わらない.

中央の E から来たものを変えるステップだけを抜き出すと，記号列の系列 E, E, I, I, I, b, b は導出 $E \Rightarrow I \Rightarrow b$ に圧縮される．この導出は，全体の導出の中で，中央の E がどのように展開されたかを正しく表している. ▨

定理 5.18 $G = (V, T, P, S)$ を CFG とし，変数 A から終端記号列 w を導く導出 $A \overset{*}{\Rightarrow} w$ があったとする．すると G に基づく再帰的推論によって，w が変数 A の言語に属していることを確かめられる.

証明 導出 $A \overset{*}{\Rightarrow} w$ のステップ数についての帰納法で証明する.

基礎：導出が 1 ステップなら，$A \rightarrow w$ が生成規則でなければならない．w は終端記号列であるから，もちろん A の言語に属している（再帰的推論手続きの第 1 歩）.

帰納：導出が $n + 1$ ステップから成り，n より少ないステップ数の導出については定理は正しいと仮定する．導出を $A \Rightarrow X_1 X_2 \cdots X_k \overset{*}{\Rightarrow} w$ とすると，前に述べたように w は $w = w_1 w_2 \cdots w_k$ の形に分解でき，次の条件が成り立つ：

a) もし X_j が終端記号ならば，$w_j = X_j$

---

2) 大きな導出からその一部分を抜きだすこの議論では，第 3 の文形式の 2 番目の変数を取り上げている．しかし以下のアイデアは，どの文形式のどの変数についてもあてはまる.

b) もし X_j が変数ならば，$X_j \stackrel{*}{\Rightarrow} w_j$ 最初のステップ $A \Rightarrow X_1X_2\cdots X_k$ はこの導出には含まれていないので，この導出のステップ数は n より小さい．したがって，帰納法の仮定から，w_j が変数 X_j の言語に属していることを再帰的推論で確かめることができる．

さて，生成規則 $A \Rightarrow X_1X_2\cdots X_k$ があり，w_j は X_j に等しいかまたは X_j の言語に属していることがわかった．再帰的推論の次の段階で，$w_1w_2\cdots w_k$ が A の言語に属していることがわかる．$w = w_1w_2\cdots w_k$ であるから，w が A の言語に属していることが確かめられた．▨

5.2.7 5.2 節の演習問題

♣ **問 5.2.1** 問 5.1.2 の文法に基づいて，各列に対する構文木を作れ．

♣ **!問 5.2.2** G を CFG とし，その生成規則の右辺はどれも空列 ε ではない，とする．もし長さ n の列 w が $L(G)$ の中にあって，m ステップの導出で得られるならば，w の構文木は $n+m$ 個の節点を持つことを示せ．

♣ **!問 5.2.3** 「生成規則の右辺が空列 ε であってもよい」ことの他は，問 5.2.2 と同じ仮定の下で，w の構文木が $n+2m-1$ 個の節点を持ち得ることと，それ以上の節点は持ち得ないことを示せ．

♣ **!問 5.2.4** 5.2.6 項で，もし $X_1X_2\cdots X_k \stackrel{*}{\Rightarrow} \alpha$ でしかも $i < j$ ならば，α の中で X_i から導かれる部分はどれも X_j から導かれる部分より左にあることを注意した．この事実を証明せよ．**ヒント**：導出のステップ数についての帰納法を使う．

5.3 文脈自由文法の応用

文脈自由文法は，自然言語を記述する手段として，チョムスキーによって考え出された．その本来の目的は達成されなかったが，コンピュータ科学で再帰的に定義される概念の利用が急増するにつれて，それらの概念の具体例を記述する手段としての CFG の必要性が高まった．ここではそのうちの二つ—— 一つは古く，一つは新しい利用例の概略を示そう．

(1) 文法はプログラミング言語の記述に使われる．重要なポイントは，CFG による言語の記述を構文解析プログラム（パーサー：コンパイラの一部で，ソース・プログラムの構造を見いだし，その構造を構文木で表現する）に変換する機械的な手順が存在することである．この応用は CFG がはじめて使われた例の一つで，コンピュータ科学の理論的なアイデアが実用と結びついた最初の応用の一つでもある．

(2)　XML（eXtensible Markup Language，インターネットで使われる言語の一つ）の発展は，発注書や製品の記述，その他たくさんの種類の文書の形式にかかわる慣習を利用者が共有できるため，電子取引がさらに使いやすくなるものと，広く予想されている．XML の本質的な部分は，**文書型定義**（DTD，Document-Type Definition）で，ここはどのようなタグがあって，どのように重ねて使えるのかを記述する，本質的に文脈自由文法である．タグとは尖った括弧で囲まれる，HTML でおなじみの `<EM>` や `</EM>` などで，強調すべきテキストを囲むものである．しかし XML のタグは，テキストの体裁ではなく，テキストの意味を扱う．たとえば電話番号と解釈してほしい文字列を，`<PHONE>` と `</PHONE>` で囲む．

5.3.1 パーサー

プログラミング言語の多くの側面は，正則表現によって記述できる構造を持っている．たとえば例 3.9 では，識別子が正則表現で表せることを示した．しかし典型的なプログラミング言語にも，正則表現だけでは表せない，非常に重要な側面もある．次にそのような例を二つ示そう．

例 5.19　典型的な言語は括弧あるいは角括弧，ときには両方を何重にも，バランスのとれた形で使用する．だからある左括弧をそのすぐ右にある右括弧に対応させて，両者を消去する——のを繰り返すことが可能でなければならない．最終的にすべての括弧が消去できれば，その列は括弧のバランスがよかったので，そうでなければ，どこかバランスが悪かったのである．バランスのとれた括弧の列の例としては，(())，()()，(()())，ε などがある．一方，)(や (() はバランスが悪い．

文法 $G_{bal} = (\{B\}, \{(,)\}, P, B)$ は，バランスのとれた括弧の列だけを，またそのすべてを生成する：ここで P は次の生成規則から成る．

$$B \to BB \mid (B) \mid \varepsilon$$

最初の規則 $B \to BB$ は，バランスのとれた二つの列をつなげば，やはりバランスのとれた列になることをいっている．それは当然で，それぞれの列で独立に括弧の対応をつければよい．その次の規則 $B \to (B)$ は，バランスのとれた列の両側を一対の括弧で囲んでも，やはりバランスのとれた列になる，といっている．これも当然で，内側の列の括弧を対応・消去してゆけば，すべて消え

失せて，先頭と末尾の括弧が隣り合うことになり，消去できる．第3の生成規則 $B \to \varepsilon$ は基礎で，空列もバランスがとれているといっている．

以上の直観的な議論でも，G_{bal} から生成される列はどれもバランスがとれていることは，納得できるであろう．その逆，つまり「バランスのとれた括弧の列は，すべて G_{bal} で生成できる」ことは証明を要する．しかしその証明は，バランスのとれた列の長さについての帰納法ででき，けっしてむずかしくないので，演習問題にしておく．

バランスのとれた列の集合は正則言語ではない，と述べたが，ここでその事実を証明しよう．もし $L(G_{bal})$ が正則言語なら，正則言語についての反復補題で保証される定数 n が，この言語にも存在する． そこでバランスのとれた列 $w = (^n)^n$，つまり n 個の左括弧の後に n 個の右括弧を並べた列を考える．反復補題にしたがって $w = xyz$ と分解すると，y は左括弧だけから成るので，xz は左括弧より右括弧のほうが多い．これはバランスが悪いので，バランスのとれた列の集合が正則であるという仮定（つまり反復補題の結論が成り立つという仮定）に矛盾する．▨

プログラミング言語はもちろん括弧以外にも多くの要素を含んでいるが，括弧は数式や条件文の本質的な部分である．図5.2の文法は，$+$ と $*$ の二つの演算しか使ってないが，数式の構造の典型的な部分である．ここには識別子の詳しい構造が扱われているが，これは3.3.2項で述べたように，コンパイラの語句解析部分で処理されることが多い．しかし図5.2で記述されている言語も，正則ではない．たとえばこの文法では，$(^n a)^n$ は合法的な式である．反復補題によれば，もしこの言語が正則であるとすると，左括弧のいくつかを削った残りの列（a と右括弧は変わらない）も合法的な式になるはず——なのにそうではない．

代表的なプログラミング言語には，バランスのとれた括弧のような性質を持つものがたくさんある．すべての型の式に含まれる括弧がそうであり，Pascal における **begin** と **end**，C における中括弧 $\{\cdots\}$ のような，プログラム区画の最初と最後もそうである．すなわち，C プログラムに含まれる中括弧は，$\{$ を左括弧，$\}$ を右括弧として「バランスのとれた列」でなければならない．

時に現れる関連するパターンで,「括弧」のバランスに「余分な左括弧があっ

てもよい」という例外が認められるものがある．C における **if** と **else** の扱いがその一例である．if 節は対応する else 節でバランスされてもよいし，else 節なしのバランスの悪い形でもよい．**if** と **else**（i と e で表す）の可能な列を生成する文法は：

$$S \to \varepsilon \mid SS \mid iS \mid iSe$$

たとえば，$ieie$，iie，iei などは **if** と **else** の可能な列であるが，これらは上の文法で生成できる．規則違反で，上の文法では生成できない列としては，ei，$ieeii$ などがある．

記号 i と e の列がこの文法で生成できるかどうかは，次のテストで簡単に判定できる（これが正しいことは，演習問題とする）：記号 e を，左から順に一つずつ検討する．注目している e の左にある，最も近い i を探す．それがなければテストは失敗で，その列は（バランスのとれた列の）言語に属していない．もし i が見つかったら，その i と注目している e とを消去する．もし e がなくなればテストは合格で，もとの列は言語に属している．e がまだ残っていたら，次の e について同じ操作を繰り返す．

例 5.20　列 iee について考える．最初の e は，すぐ左の i に対応している．これらを消去すると，e だけが残る．残りの e があるので，それについて同じ操作を適用すると，それより左には i がない．したがってテストは失敗で，iee は言語に属していない．C プログラムで **if** より **else** のほうが多いことはあり得ないので，この結論は正しい．

```
if (条件) {
    ...
    if (条件) 文;
    else 文;
    ...
    if (条件) 文;
    else 文;
    ...
}
```

図 5.10　if-else 構造の例：二つの **else** が直前の **if** と組になっていて，最初の **if** は単独で使われている．

もう一つの例として，$iieie$ を考えてみよう．最初の e とその左の i を消去すると，iie が残る．残りの e とその左の i を消去すると，i が残る．e は残っていないので，テストは成功である．この結論も妥当で，列 $iieie$ は，図 5.10 のような構造の C プログラムに対応している．実際，i と e を対応づける手順は，我々に（C コンパイラにも）どの **if** がどの **else** と組になるかを教えてくれる．この事実はコンパイラがプログラマの意図する計算の流れを組み立てるときに本質的に役立つ．▨

5.3.2 YACC パーサー生成プログラム（Parser-Generator）

与えられたソース・プログラムから構文木を構成する機能を持つパーサーを生成する作業は，すべての UNIX システムにある YACC コマンドに組織化されている．YACC への入力は CFG で，その記法はここで使われているものと細かいところしか違わない．各生成規則に**アクション**と呼ばれる C プログラムが結びつけられ，その生成規則に対応する構文木の節点（とその子）が作られるたびに実行される．ふつうはアクションとは節点を作り出すプログラムであるが，ある YACC システムではアクションは木を実際に作るのではなく，オブジェクト・プログラムの一部分を出力するなど，他の仕事をする．

例 5.21 図 5.11 は YACC 記法で表した CFG の見本で，図 5.2 と同じ文法が扱われている．ここではアクションを省略し，ただ YACC への入力の中でのそれらの位置を中括弧で示した．

YACC 記法による文法の記述と我々の記法との対応関係は次の通りである．

- 我々の生成記号 → を，コロン（:）で表す．
- 生成規則は頭部ごとにまとめて，本体をタテ線（|）で区切って並べる．我々もこの記法を使うことがある．
- 特定の頭部に対する本体のリストは，セミコロン（;）で区切る．この区切り記号は我々は使っていない．
- 終端記号は一重の引用符（'）で囲む．同じ囲みの中に複数個の終端記号を書いてもよい．ここでは示さなかったが，YACC は新しい終端記号を利用者が定義することもできる．ソース・プログラムにそのような終端記号が現れると，字句解析プログラムがそれを検出して，その返り値によって，パー

```
Exp : Id              {...}
    | Exp '+' Exp {...}
    | Exp '*' Exp {...}
    | '(' Exp ')' {...}
    ;
Id  : 'a'             {...}
    | 'b'             {...}
    | Id 'a'          {...}
    | Id 'b'          {...}
    | Id '0'          {...}
    | Id '1'          {...}
    ;
```

図 5.11　YACC 記法で表した CFG の例

サーに通報する．

- 引用符なしで書かれている文字は変数名である．これを利用して，E や I を使ってもよかったのであるが，もっとわかりやすい名前——**Exp** と **Id** を使った．▨

5.3.3 マークアップ言語

次に「マークアップ言語」と呼ばれる言語について考えよう．それらの言語の「列」は，文書にあるマーク（**タグ**と呼ばれる）を埋め込んだものである．タグは文書内のいろいろな文字列の意味について，何かを知らせてくれる．

マークアップ言語で最もよく知られているのは HTML（Hyper Text Markup Language）であろう．この言語は二つの主要な働きを持っている：文書の間のリンクを張ることと，文書の体裁（「見かけ」）を記述することである．ここでは HTML の構造を単純化したものだけ紹介するが，次の例からその構造と，CFG がどのように合法的な HTML 文書を記述しその文書の処理（ディスプレイあるいはプリンタへの出力）を助けるかがわかるであろう．

例 5.22　図 5.12 (a) は，項目のリストを含むテキストの断片を示し，図 5.12 (b) はその HTML 表現を示している．組を作るタグは，ある列 x について `<x>`

The things I *hate*:

1. Moldy bread.
2. People who drive too slow in the fast lane.

(a) 表示される形

```
<P>The things I <EM>hate</EM>:
<OL>
<LI>Moldy bread.
<LI>People who drive too slow
in the fast lane.
</OL>
```

(b) HTML ソース

図 5.12 HTML 文書 (b) とそれを印字した結果 (a)

と </x> という形をしている．<EM> と </EM> がその例で，その間のテキストを強調することを指示している．<OL> と </OL> も組を作るタグで，順序付きのリスト，すなわちリストの項目を列挙することを指示している．

他に組を作らないタグの例が二つ現れている：<P> と <LI> である．これらはそれぞれ，パラグラフとリストの項目の始まりを示している．HTML はこれらのタグと組み合わせてパラグラフや項目の終わりに </P> や </LI> をおくことも許し，むしろ奨励しているが，単独で使うことも許されている．▨

HTML 文書に結びつけられている列のクラスがいくつかある．それらのすべてを列挙することはしないが，例 5.22 ののようなテキストを理解するのに必要なものを次に挙げておこう．どのクラスでも，わかりやすい名前の変数を導入する．

(1) *Text*（テキスト）とは文字通りに解釈される，つまりタグを含まない，任意の記号列のことである．図 5.12 (a) では，“Moldy bread.” が *Text* の一例である．

(2) *Char*（記号）とは，HTML テキストで使える，1 記号だけから成る列のことである．なお空白も記号に含まれる．

(3) *Doc*（文書）は文書，すなわち “element”（要素）の列を表す．要素はこの後で，*Doc* とあわせて相互再帰的に定義される．

(4) *Element*（要素）とは，*Text* 列であるか，文書を間に挟むタグの組，あるいは単独のタグのあとに文書をつけたものをいう．

(5)　*ListItem*（リスト項目）とは，タグ <LI> のあとに，リストの一つの項目を表す文書をつけたものである．

(6)　*List*（リスト）とは 0 個以上のリスト項目の列のことである．

(1)	*Char*	$\rightarrow$	$a \mid A \mid \cdots$
(2)	*Text*	$\rightarrow$	$\varepsilon \mid$ *Char Text*
(3)	*Doc*	$\rightarrow$	$\varepsilon \mid$ *Element Doc*
(4)	*Element*	$\rightarrow$	*Text* |
			<EM> *Doc* </EM> |
			<P> *Doc* |
			<OL> *List* </OL> | ···
(5)	*ListItem*	$\rightarrow$	<LI> *Doc*
(6)	*List*	$\rightarrow$	$\varepsilon \mid$ *ListItem List*

図 5.13　HTML 文法の一部

図 5.13 は HTML 言語の構造のうち，これまでに使用した範囲を記述する CFG である．第 1 行目（1）では，記号には “a” や “A” などの他の多くの文字・記号があり，それらが HTML の記号セットを作っていることが示されている．第 2 行（2）では二つの生成規則によって，*Text* が空列かまたは使用できる記号のあとにさらにテキストをつけ加えたものであることをいっている．いい方を変えれば，*Text* とは 0 個以上の列のことである．なお記号 <，>—はテキストの中では許されないが，それらを &lt と &gt で表すことができる．したがって「テキストの中に，タグが偶然にまぎれこむ」ことはあり得ない．

第 3 行（3）は，文書が 0 個以上の要素の列であることをいっている．そして要素とは，第 4 行（4）で示されているように，テキストであるか，強調されたテキストであるか，またはパラグラフの先頭記号に文書をつけたもの，あるいはリスト，などである．なおここでは使われてない HTML タグに対応する，要素を展開するための生成規則もある．また第 5 行（5）ではリスト項目とはタグのあとに文書をつけたものであること，第 6 行（6）ではリストとは 0 個以上のリスト項目であることを述べている．

HTML のいくつかの面は，文脈自由文法の力を借りなくても記述できる；正則表現で十分である．たとえば図 5.13 の（1），（2）は，*Text* が正則表現

$(\mathbf{a}+\mathbf{A}+\cdots)^*$ と同じ言語を表すことしかいっていない．しかし，HTML の別の面では，CFG の力を必要とする．たとえば `<EM>` と `</EM>` のような，開始記号と終了記号の対がバランスのよい括弧構造をなすことは，すでに学んだように正則表現では表せない．

5.3.4 XML と文書型の定義

HTML が文法で記述されるという事実は，特に注目すべきことではない．本質的に，すべてのプログラミング言語はそれぞれの CFG で記述できるので，もし HTML がそのように記述できないとしたら，それこそ驚くべきことである．しかし，もう一つの重要なマークアップ言語 XML を見ると，この言語を使用する過程の一部で，CFG がさらに重要な役割を果たしていることに気づく．

XML の目的は，文書の体裁を指定することではない；それは HTML の仕事である．それよりは，XML はテキストの「意味内容」を記述しようとする．たとえば,「銀座通り 12」は番地のように見えるが，ほんとうにそうなのだろうか？ XML では番地を表す語句を囲むタグが使える；たとえば：

```
<ADDR>銀座通り 12</ADDR>
```

しかし `<ADDR>` が建物の番地を表すかどうか，これだけでは定かでない．文書が記憶装置の割り当てを扱っている場合には，タグ `<ADDR>` は記憶装置の番地と解釈されるであろう．どんなタグがありタグのどんな組み合わせがどのような構造を作り出すかを明確にするには，共通の関心を持つ人々が標準規格を DTD（Document-Type Definition，文書型定義）の形で開発することが望まれる．

DTD は，変数や生成規則を表す記法は異なるが，本質的に文脈自由文法である．次の例で，簡単な DTD を示し，DTD を記述するための言語を導入しよう．DTD 言語はそれ自体文脈自由文法であるが，ここで説明したいのはその文法ではない．それよりは，DTD を記述する言語は本質的に CFG 記法なので，どのようにして CFG をその言語で記述するのかを明らかにしたい．

DTD の形式は次の通りである：

```
<!DOCTYPE DTD の名前 [
        要素の定義のリスト
```

```
]>
```

そして要素の定義は

```
<!ELEMENT 要素の名前 (要素の記述)>
```

という形式で,「要素の記述」は本質的に正則表現である．その表現の基本は：

(1)　他の要素の名前 … 一つの型の要素が他の型の要素に混じって現れるという事実を表現する．これは HTML でリストの中に強調されたテキストが見られるのと同じことである．

(2)　特別の項 `\#PCDATA` … これは XML のタグに関係しない，任意のテキストを表す．これは例 5.22 の変数 *Text* と同じ役割を担っている．

使用できる演算記号は：

(1)　タテ棒（|）は，3.3.1 項で紹介した UNIX の正則表現と同じで，和（「または」）を表す．

(2)　コンマ（,）は連接（列をつなぐこと）を表す．

(3)　3.3.1 項で述べたいわゆる閉包演算の変種：「0 個以上の列」を表す通常の演算 “*”,「1 個以上の列」を表す “+”，それから「0 個か 1 個の出現」を表す “?” がある．

括弧は演算記号とその対象となる文字・記号のグループ分けに使われる；その他，正則表現の演算記号の通常の優先順位が適用される．

例 5.23　コンピュータの売り手が，彼らが今販売しているいろいろなパソコンの紹介を，ウェブ上で出版するのに使えるような，標準的な DTD を作るために集まったとしてみよう．パソコンの紹介は，モデル番号と，そのモデルの特徴——RAM の大きさやディスクの数と容量など，詳しい情報を含む．図 5.14 は仮想的な，非常に簡単なパソコン用の DTD を示している．

この DTD の名前は `PcSpecs` である．最初の要素は，CFG の開始記号のようなものであるが，`PCS` である．これはパソコンの仕様のリストであって，その定義 `PC*` は「PCS が 0 個以上の `PC` 項目の列である」ことを示している．

次に要素 `PC` の定義が示されている．PC は五つのものの連接で，最初の四つは他の要素，すなわちモデル（MODEL），価格（PRICE），処理装置の型（PROCESSOR），それにパソコンの主記憶装置（RAM）である．コンマ（,）

```
<!DOCTYPE PcSpecs [
   <!ELEMENT  PCS (PC*)>
   <!ELEMENT  PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
   <!ELEMENT  MODEL (\#PCDATA)>
   <!ELEMENT  PRICE (\#PCDATA)>
   <!ELEMENT  PROCESSOR (MANF, MODEL, SPEED)>
   <!ELEMENT  MANF (\#PCDATA)>
   <!ELEMENT  MODEL (\#PCDATA)>
   <!ELEMENT  SPEED (\#PCDATA)>
   <!ELEMENT  RAM (\#PCDATA)>
   <!ELEMENT  DISK (HARDDISK | CD | DVD)>
   <!ELEMENT  HARDDISK (MANF, MODEL, SIZE)>
   <!ELEMENT  SIZE (\#PCDATA)>
   <!ELEMENT  CD (SPEED)>
   <!ELEMENT  DVD (SPEED)>
]>
```

図 5.14　パソコンのための DTD

は連接を表すので，これらの要素はその順に 1 回ずつ現れないといけない．最後の成分 DISK+ は，そのパソコンのディスクについての項目が 1 個以上続くことを示している．

多くの成分は単純なテキスト（文字列）である：MODEL，PRICE，RAM はこの型である．しかし PROCESSOR は，下部構造を持っている．その定義から，それが製造者（MANF，manufacturer），モデル，速度から成り，これらの成分がその順に並んでいることがわかる；どの成分も単純なテキストである．

DISK 項目はさらに複雑である．まず，ディスクとはハードディスクか，CD，あるいは DVD である——これは，要素ディスク（DISK）の定義で，三つの他の要素が「または」で結ばれていることからわかる．ハードディスクについては，製造者・モデル・容量が指定されるが，CD と DVD は速度だけで表現される．

図 5.15 は図 5.14 の DTD に適合する XML 文書の例である．この文書ではどの要素も要素の名前を持つタグで表現され，終わりに対応するタグ（HTML

```
<PCS>
   <PC>
      <MODEL>4560</MODEL>
      <PRICE>$2295</PRICE>
      <PROCESSOR>
         <MANF>Intel</MANF>
         <MODEL>Pentium</MODEL>
         <SPEED>800MHz</SPEED>
      </PROCESSOR>
      <RAM>256</RAM>
      <DISK><HARDDISK>
         <MANF>Maxtor</MANF>
         <MODEL>Diamond</MODEL>
         <SIZE>30.5Gb</SIZE>
      </HARDDISK></DISK>
      <DISK><CD>
         <SPEED>32x</SPEED>
      </CD></DISK>
   </PC>
   <PC>
      ...
   </PC>
</PCS>
```

図 5.15　図 5.14 の DTD の構造にしたがう，文書の一部分

の場合のように，斜線付きのタグ）がつけられている．だから図 5.15 では，最も外側のタグは <PCS>...</PCS> で，これらのタグの内側に，この製造者によって販売されているパソコンごとに，一つずつ項目が並べられている；ただし図では一つしか明示していない．

例示されている <PC> 項目を見れば，モデル番号が 4560 であり，価格が 2295 ドルであることがすぐわかる．処理装置はインテルの 800 メガヘルツのインテル・ペンティアム・プロセッサである．主記憶装置は 256 メガバイトで，30.5 ギガバイトの Maxtor Diamond ハードディスクがついている．重要なのは，我々

がこのような情報を読みとれることではなく，プログラムが文書を読みとり，図5.14のDTDをも読みとった上でそれにしたがって，図5.15の数や名前を正しく解釈（して画面に表示）できることである．▨

図5.14のようなDTDの要素に対する規則は，文脈自由文法の生成規則とはまったく同じではない．しかし多くは標準に合った形をしている．たとえば

```
<!ELEMENT  PROCESSOR(MANF, MODEL, SPEED)>
```

は生成規則

$$Processor \rightarrow Manf\ Model\ Speed$$

にちょうど対応している．一方，規則

```
<!ELEMENT  DISK(HARDDISK | CD | DVD)>
```

は DISK を具体的に展開する定義を明示していない．この場合は，展開は簡単である：この規則は，次の三つの規則と解釈すればよい．

$$Disk \rightarrow Harddisk \mid Cd \mid Dvd$$

ここでタテ棒（|）は共通の頭部を持つ生成規則の略記法という，前と同じ意味で使われている．

もっとむずかしいのは

```
<!ELEMENT  PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

の場合である．ここでは「本体」の中で演算記号＋が使われている．これは DISK+ を新しい変数，たとえば *Disks* に置き換えて，1個以上の変数 *Disk* を生成できるようにすれば解決できる．これと等価な生成規則は：

$$Pc \rightarrow Model\ Price\ Processor\ Ram\ Disks$$

$$Disks \rightarrow Disk \mid Disk\ Disks$$

生成規則の本体に正則表現を含む CFG を，標準的な CFG に変換する一般的な技法がある．ここではその方針を直観的に述べておこう；生成規則に正則表現を含む CFG の意味を定式化し，CFL を越える新しい言語にはならないことを証明したい．それにはまず正則表現を含む生成規則を一つずつ，それと等価な通常の生成規則群に置き換えられることを帰納法で示す．帰納法は，本体

に含まれる正則表現の長さについて行う．

基礎：本体が要素の連接であれば，規則はそのままでCFGの合法的な形になっている．だから何もしなくてよい．

帰納：さもなければ，最後の演算記号に応じて，次の五つの場合に分けて考える．

(1)　規則が $A \to E_1, E_2$ という形であるとき：ここではDTD言語で許される表現とする．これは最後の演算が連接（,）の場合である．この場合は B，C という変数を導入して，規則 $A \to E_1, E_2$ を次の生成規則の置き換えればよい．

$$A \to BC$$

$$B \to E_1$$

$$C \to E_2$$

最初の生成規則 $A \to BC$ は，CFGで許される形である．他の二つは，許される形かどうかわからないが，もとの規則の本体より短いので，帰納法の仮定により標準的なCFGの形に変換できる．

(2)　規則が $A \to E_1 \mid E_2$ という形をしているとき：この「合併」演算については，規則を

$$A \to E_1$$

$$A \to E_2$$

に分解すればよい．これらがCFGの合法的な規則であるかどうかはわからないが，どちらの本体ももとの本体より短いので，再び帰納法の仮定によって，CFGの標準的な規則に変換できる．

(3)　規則が $A \to (E_1)^*$ という形であるとき：他で使われていない新しい変数 B を導入して，次の生成規則に置き換えればよい．

$$A \to BA$$

$$A \to \varepsilon$$

$$B \to E_1$$

(4)　規則が $A \to (E_1)^+$ という形であるとき：他で使われていない新しい変数 B を導入して，次の生成規則に置き換えればよい．

$$A \to BA$$

$$A \to B$$

$$B \to E_1$$

(5)　規則が $A \to (E_1)?$ という形であるとき：次の生成規則に置き換えればよい．

$$A \to \varepsilon$$

$$A \to E_1$$

例 5.24　DTD の規則

```
<!ELEMENT  PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

を合法的な CFG の規則に変換してみよう．まず，この規則の本体を二つの表現の連接とみなす：最初は MODEL, PRICE, PROCESSOR, RAM で，第 2 は DISK+ である．これらの部分表現を変数，たとえばで表せば，次の生成規則が得られる：

$$Pc \to AB$$

$$A \to Model\ Price\ Processor\ Ram$$

$$B \to Disk^{+}$$

最後の規則だけが，合法的でない形をしている．そこで新しい変数 C と，次の生成規則を導入する：

$$B \to CB \mid C$$

$$C \to Disk$$

これは特殊な場合で，表現 A はいくつかの変数の単なる連接であり，$Disk$ は一つの変数である．だから実は，変数 A，C は使わなくてもすむ．だから次の規則でもよい：

$$Pc \to Model\ Price\ Processor\ Ram\ B$$

$$B \to DiskB \mid Disk$$ ▨

5.3.5　5.3 節の演習問題

♣ **問 5.3.1**　括弧の列のバランスが（例 5.19 で述べた意味で）よければ，文法 $B \to BB\,|\,(B)\,|\,\varepsilon$ で生成されることを証明せよ．

ヒント：列の長さについての帰納法を使え．

♣ ***問 5.3.2**　二つの型の括弧——丸括弧 ( ) と角括弧 [ ]——のバランスのとれた列を考える．これらの列は，次のようなところで現れる．C の数式では，丸括弧は式のまとめや関数の引数を囲むのに使われ，角括弧は配列の添え字を表すのに使われる．そこでこれらの括弧以外をすべて消すと，二つの型の括弧の，バランスのとれた列が残る．たとえば

```
f(a[i]*(b[i][j],c[g(x)]),d[i])
```

からは，バランスのとれた列 `([]([][][()])[])` が得られる．このような「丸括弧と角括弧のバランスのとれた列」のすべて，またそれらだけを生成する文法を設計せよ．

♣ **!問 5.3.3**　5.3.1 項で，文法

$$S \to \varepsilon\,|\,SS\,|\,iS\,|\,iSe$$

について，ある列 w がこの言語 L に属しているかどうかは，次のことからわかる，といった．

a)　w の先頭が e であったら，失敗：w は L に属してない．

b)　w が e を含んでいなければ（i はあってもよい），成功：w は L に属している．

c)　その他の場合は，最初の e と，そのすぐ左の i を消す．そして新しい列（同じ w で表す）について，上のテストを繰り返す．

この手順で L の中の列が正しく識別できることを証明せよ．

♣ **問 5.3.4**　図 5.13 の HTML 文法に，次の形式をつけ加えよ：

*a)　リストの項目の最後は閉じるタグ `</LI>` でなければならない．

b)　要素は「順序のないリスト」であってもよい．順序のないリストはタグ `<UL>` とそれを閉じる `</UL>` で囲まれる．

!c)　要素は表であってもよい．表は `<TABLE>` と閉じ役の `</TABLE>` で囲まれる．これらのタグの間に 1 個以上の行が並び，どの行も `<TR>` と `</TR>` で囲まれる．最初の行は見出しで，いくつかの欄を含み，それらはタグ `<TH>` で始まる（閉じるべきであるが，ここでは閉じないと仮定する）．それに続く行は，タグ `<TD>` で始まる．

♣ **問 5.3.5**　図 5.16 の DTD を文脈自由文法に変換せよ．

```
<!DOCTYPE  CourseSpecs [
<!ELEMENT  COURSES (COURSE+)>
<!ELEMENT  COURSE (CNAME, PROF, STUDENT*, TA?)>
<!ELEMENT  CNAME (\#PCDATA)>
<!ELEMENT  STUDENT (\#PCDATA)>
<!ELEMENT  TA (\#PCDATA)>
]>
```

図 5.16　授業科目に対する DTD

5.4　文法と言語のあいまいさ

すでに触れたように CFG の応用は，多くの場合，文法がファイルの構造を示すところにある．たとえば 5.3 節では，文法がプログラムと文書の構造を規定するのにどのように役立つかを示した．そこでの暗黙の了解は，それぞれの言語に属する列の構造が，文法によって一義的に決定される，ということであった．しかしあとでわかるように，すべての文法が一義的な構造を提供するとは限らない．

ある文法が一義的な構造を提供できなかったとき，その文法に手を加えて，対象とする言語の各列の構造を一義的に定められるように修正できる場合がある．しかし残念ながら，それができない場合もある．すなわち，ある CFL は「本質的にあいまい」であって，その言語に対するどんな CFG も，その言語のある列に対して複数個の構造を定める．

5.4.1　あいまいな文法

いつもの例：図 5.2 の文法に戻ろう．この文法は，演算記号 $*$，$+$ の任意の列を含む数式を生成する．そして生成規則 $E \to E+E \mid E*E$ のもとでは，それらの式を好き勝手な順序で生成することができる．

例 5.25　たとえば，文形式 $E+E*E$ を考えてみよう．これは E から，次のように 2 通りのしかたで導出できる：

(1)　$E \Rightarrow E+E \Rightarrow E+E*E$

(2)　$E \Rightarrow E*E \Rightarrow E+E*E$

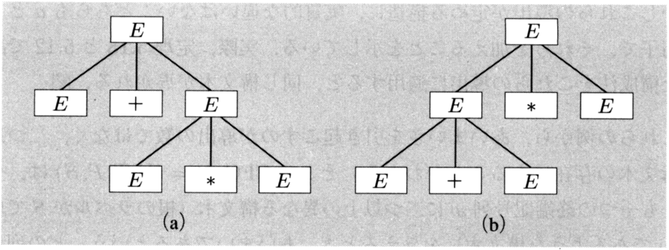


図 5.17　同じ成果の，二つの構文木

最初の導出では，第 2 の E が $E*E$ に置き換えられ，次の導出では，第 1 の E が $E+E$ に置き換えられている．図 5.17 はそれらに対応する構文木を示しているが，それらが異なることに注意してほしい．

これらの導出の違いは重要である．式の構造に関する限り，(1) は第 2 と第 3 の式が掛けられ，その結果が第 1 の式に加えられるのに，(2) では最初の二つの式を加え，その結果に第 3 の式を掛けている．もっと具体的にいうと，最初の導出は $1+2*3$ を $1+(2*3)=7$ のようにグループ分けするのに，次の導出では同じ式を $(1+2)*3=9$ のようにグループ分けしている．明らかに，計算式のグループ分けの慣習（乗除算は加減算に優先する）に合致するのは前者であって，後者ではない．

図 5.2 の文法は，$E+E*E$ の中の式 E を識別子に置き換えて得られるどんな終端記号列にも二つの異なる構造を与えるので，一義的な構造を与えるという点ではよい文法とはいえない．特に，数式によいグルーピングを与えることもあるが，正しくないグルーピングを行うこともある．この文法をコンパイラで使うときは，正しいグルーピングだけを行うような修正をしなければならない．▨

一方，一つの列に（異なる構文木ではなく）異なる導出がある，というだけなら，文法の欠陥にはならない．次のその例を挙げよう．

例 5.26　同じ式の文法を使うと，列にたくさんの異なる導出がある．二つの例を挙げると：

(1)　$E \Rightarrow E+E \Rightarrow I+E \Rightarrow a+E \Rightarrow a+I \Rightarrow a+b$

(2)　$E \Rightarrow E+E \Rightarrow E+I \Rightarrow I+I \Rightarrow I+b \Rightarrow a+b$

しかしこれらの導出が定める構造に，実質的な違いはない：どちらも a と b が識別子で，それらを加えることを示している．実際，定理 5.18 と 5.12 で説明した構成法をこれらの導出に適用すると，同じ構文木が導かれる．▨

これらの例から，あいまいさを引き起こすのが導出の数ではなく，二つ以上の構文木の存在にあることがわかる．そこで CFG $G = (V, T, P, S)$ は，少なくとも一つの終端記号列 w に二つ以上の異なる構文木（根のラベルが S で成果が w であるような構文木）を与えるとき，**あいまいである**という．どの列にも構文木が高々一つしかないとき，その文法は**あいまいでない**という．

たとえば例 5.25 は，図 5.2 の文法があいまいであることの，ほとんど証明になっている——あとは図 5.17 の木を終端記号列を成果とする木として完成させられることさえ示せばよい．図 5.18 はその「完成」の例である．

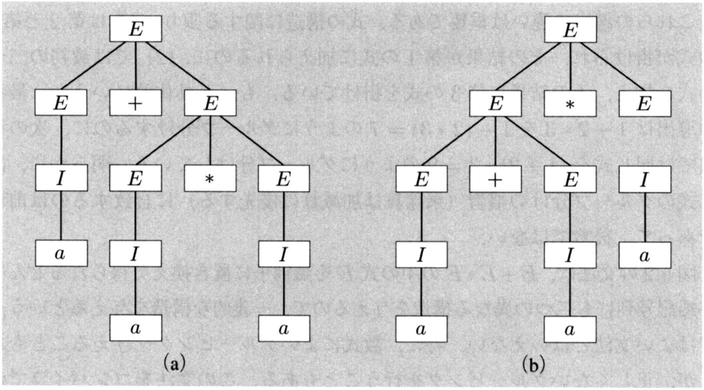


図 5.18　$a + a * a$ を成果とする木（式の文法があいまいであることを示す）

5.4.2　文法のあいまいさの除去

理想的な世界では，4.4 節でオートマトンのムダな状態を除去するアルゴリズムを示したように，CFG のあいまいさを除去するアルゴリズムを提供できるであろう．しかし驚くべきことに，9.5.2 項で証明することであるが，与えられた CFG が，そもそもあいまいであるか否かを判定できるアルゴリズムは存在しない．さらに，5.4.4 項で示すように，文脈自由言語で，あいまいな CFG し

YACCでのあいまいさ対策

使っている式の文法があいまいなら，図5.11のYACCプログラムが現実的かどうか，疑いが生じる．基礎となる文法があいまいなのは事実であるが，YACCのパーサー生成能力は，あいまいさをひき起こす原因のほとんどを解決する単純な機構を利用者に提供してくれる．式の文法については，次の点を指摘すれば十分である：

a) $*$ は $+$ より優先順位が高い．すなわち $*$ は，どちらかの側に隣接する $+$ よりも先にグループにされる（実行される）．この規則は，例5.25では導出（1）を採用し，(2) を選ばないことを意味する．

b) $*$ も $+$ も，左結合的である．すなわち，$*$ だけでつながれた式の列は，左から順にグループにする（実行する）．$+$ についても同様である．

YACCでは，演算の優先順位を，それらを順位の最も低いものから高いものへと並べることによって，指定できる．技術的には演算の優先順位は，関連する演算が本体の右端の終端記号であるような生成規則を使うときに関わってくる．演算が左あるいは右結合的であることも，キーワード **%left** あるいは **%right** によって宣言できる．たとえば，$+$ と $*$ がどちらも左結合的であって，$*$ のほうが $+$ より優先順位が高いことは，図5.11の文法の前に次のような文をおくことによって宣言できる：

```
%left  '+'
%left  '*'
```

か存在しないものがある．そのような言語の文法の，あいまいさを除去するのは不可能である．

幸いなことに，実際的な状況はそれほど深刻ではない．ふつうのプログラミング言語で現れる種類のあいまいさに対しては，それらを除去するよく知られた技法がある．図5.2の式の文法にまつわる問題が典型的なので，そのあいまいさの除去を重要な例として取り上げてみよう．

まず，図5.2の文法のあいまいさには，二つの原因があることを注意しておこう．

(1) 演算の優先順位が考慮されていない．図5.17 (a) は $*$ を $+$ より先に，正しくグループにしている一方，同図 (b) は構文木ではあるが，$+$ を $*$ より先にグループにしている．あいまいでない文法においては，図5.17 (a) だけが合法的な構造となるようにする必要がある．

(2) 同じ演算の列は左からグループにしても右からにしてもよい．たとえば，図 5.17 の $*$ を $+$ に置き換えると，列 $E+E+E$ の二つの異なる構文木が得られる．しかし加法も乗法も結合法則を満たすので，グループ化が左からでも右からでも答は同じになる．しかしあいまいさを消すためには，どちらか一つを選ばなければならない．ふつうは左からグループにしてゆくので，図 5.17 (b) が二つの $+$ に対する唯一の正しいグループ化である．

優先順位を定める問題は,「結合の強さ」が同じ水準にある式を表す変数をいくつか導入すれば解決できる．具体的にいうと：

(1) **因数**とは隣接する演算記号 $*$ や $+$ によって分割されない式のことである．我々が今扱っている数式の場合，次のものだけが因数である．

a) 識別子：識別子に演算記号をつけても識別子を分割することはできない．

b) 括弧で囲まれた式：括弧の内側はどんな式でもよい．括弧はその中にあるものを，括弧の外の演算から切り離すために使うものである．

(2) **項**とは，演算+によって分割されない式のことである．我々の場合，演算は $+$ と $*$ だけなので，項とは一つ以上の因数の積のことである．たとえば $a*b$ は，左結合性にしたがう場合，その左に $a1*$ をおくと「分割」される——$a1*a*b$ は，$(a1*a)*b$ とグループ化される．しかし左に $a1+$ をつけても右に $+a1$ をおいても，$a*b$ は分割されない．$a1+a*b$ の正しいグループ分けは $a1+(a*b)$ であり，$a*b+a1$ の正しいグループ分けは $(a*b)+a1$ である．

(3) **式**とは，隣接する $+$ や $*$ によって分割され得るものをも含む，任意の式を表す．したがって我々の場合，式とは 1 個以上の項の和を意味する．

例 5.27 図 5.19 は図 5.2 と同じ言語を生成する，あいまいでない文法を示す．F, T および E を，上に定義した因数，項，式を言語とする変数と考える．たとえば，この文法は列 $a+a*a$ に対して，図 5.20 に示すただ一つの構文木を与える．▨

この文法があいまいでないというのは，明らかとはとてもいえない．次に，この言語のどんな列でも「二つの異なる構文木を持つことはあり得ない」理由を

$$I \to a \,|\, b \,|\, Ia \,|\, Ib \,|\, I0 \,|\, I1$$
$$F \to I \,|\, (E)$$
$$T \to F \,|\, T * F$$
$$E \to T \,|\, E + T$$

図 5.19　数式の，あいまいでない文法

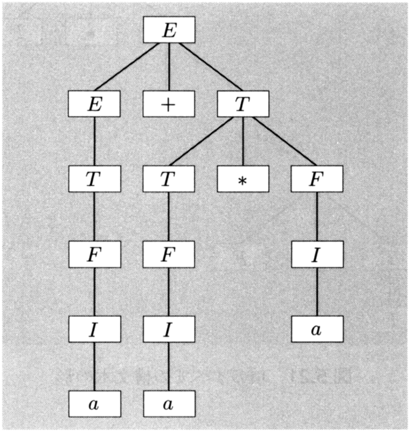


図 5.20　$a + a * a$ のただ一つの構文木

説明する，基本的な観察を述べておこう．

- T から導かれる列，項は，$*$ でつながれたいくつかの因数の列である．因数とは，上の定義と図 5.19 の文法からわかるように，一つの識別子であるかまたは括弧で囲まれた式のことである．
- T に対する二つの生成規則の形からわかるように，因数の列に対する構文木はただ一つで，$f_1 * f_2 * \cdots * f_n$ $(n > 1)$ を項 $f_1 * f_2 * \cdots * f_{n-1}$ と因数 f_n の積に分解する．その理由は，F から積 $f_{n-1} * f_n$ は（その外側に括弧をつけない限り）導かれないからである．だから生成規則 $T \to T * F$ を使うとき，F からは最後の因数しか導かれない．すなわち，項に対する

構文木は図 5.21 のようなものに限る．

- 同じように，式は項を + でつないだものである．生成規則 $E \to E+T$ によって式 $t_1+t_2+\cdots+t_n$ を導いたとき，T からは t_n しか導かれないし，本体の E からは $t_1+t_2+\cdots+t_{n-1}$ が導かれる．その理由はやはり，T からは二つ以上の項は，括弧で囲まない限り導かれないからである．

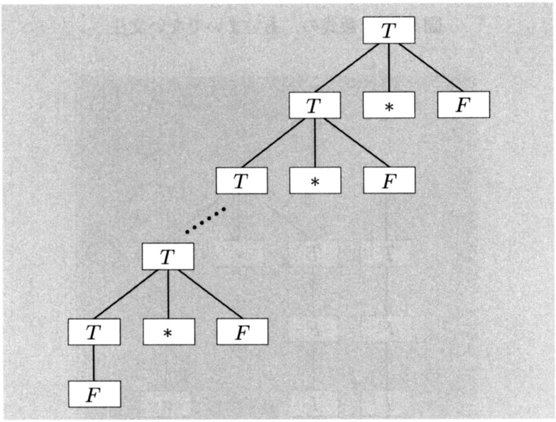


図 5.21 項のすべての構文木の形

5.4.3 あいまいさを表現する手段としての最左導出

文法があいまいでないとしても，導出はただ一つとは限らないが，あいまいでない文法では，最左導出はただ一つで，最右導出もただ一つである．ここでは最左導出だけを考えるが，結果は最右導出についても成り立つ．

例 5.28 $E+E*E$ を成果とする，図 5.18 の二つの構文木に注目してみよう．それらの最左導出は次のようになる：

a) $E \underset{左}{\Rightarrow} E+E \underset{左}{\Rightarrow} I+E \underset{左}{\Rightarrow} a+E \underset{左}{\Rightarrow} a+E*E \underset{左}{\Rightarrow}$
$a+I*E \underset{左}{\Rightarrow} a+a*E \underset{左}{\Rightarrow} a+a*I \underset{左}{\Rightarrow} a+a*a$

b) $E \underset{左}{\Rightarrow} E*E \underset{左}{\Rightarrow} E+E*E \underset{左}{\Rightarrow} I+E*E \underset{左}{\Rightarrow} a+E*E \underset{左}{\Rightarrow}$
$a+I*E \underset{左}{\Rightarrow} a+a*E \underset{左}{\Rightarrow} a+a*I \underset{左}{\Rightarrow} a+a*a$

これらの最左導出が異なることに注意しよう．これは定理の証明にはならないが，木の相違が最左導出の相違にどのように反映されるかを示している．▨

定理 5.29 文法 $G = (V, T, P, S)$ と終端記号列 w に対して，w が二つの異なる構文木を持つ必要十分条件は，S から w を導く二つの異なる最左導出が存在することである．

証明 （必要性）定理 5.14 の証明で述べた「構文木から最左導出を構成する方法」を吟味してみよう．二つの構文木で異なる生成規則が適用される最初の節点に注目すると，構成される最左導出でも，そこで異なる生成規則が適用されるはずである．
（十分性）これまで最左導出から構文木を構成する直接的な手順は示していなかったが，それほどむずかしいことではない．まずラベル S を持つ根だけから成る木を作ることから始める．導出を 1 ステップずつ調べる．各ステップで，ある変数が置き換えられるが，それは構成中の木で，その変数をラベルとする，子がない左端の節点に対応している．最左導出のそのステップで使われている生成規則から，その節点の子が定まる．もし二つの異なる最左導出があるなら，それらが異なる最初のステップで，対応する節点に異なる子のリストが指定される．したがって，構成される構文木は明らかに異なる．▨

5.4.4 本質的なあいまいさ

文脈自由言語 L は，その文法がすべてあいまいであるとき，**本質的にあいまいである**といわれる．L のあいまいでない文法が一つでもあれば，L は言語としてあいまいでない．たとえば例 5.2 の文法で生成される式の言語はあいまいでない．その文法はあいまいであるが，他のあいまいでない文法——図 5.19 の文法によって同じ言語が生成されるからである．

本質的にあいまいな言語が存在することの証明はしない．それより，本質的にあいまいである言語の一例を取り上げて，それを生成するどんな文法もあいまいであることを，直観的に説明してみよう．その言語 L とは：

$$L = \{a^n b^n c^m d^m \mid n \geq 1,\, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1,\, m \geq 1\}$$

すなわち，L は $\mathbf{a}^+\mathbf{b}^+\mathbf{c}^+\mathbf{d}^+$ に属する列で，次の条件のどちらかを満たすものから成る：

(1) a と b は同数で，c と d は同数である．

(2) a と d は同数で，b と c は同数である．

$$S \to AB \mid C$$
$$A \to aAb \mid ab$$
$$B \to cBd \mid cd$$
$$C \to aCd \mid aDd$$
$$D \to bDc \mid bc$$

図 5.22　本質的にあいまいな言語を生成する文法

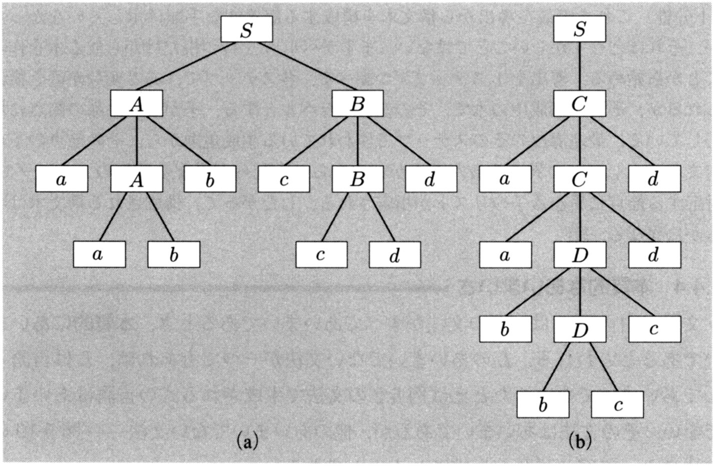


図 5.23　$aabbccdd$ の二つの構文木

L は文脈自由言語で，図 5.22 で示す文法によって生成される．その文法は，L の 2 種類の列を生成する，独立の規則群を使用する．

この文法はあいまいである．たとえば，列 $aabbccdd$ は次の二つの最左導出を持つ：

(1)　$S \underset{左}{\Rightarrow} AB \underset{左}{\Rightarrow} aAbB \underset{左}{\Rightarrow} aabbB \underset{左}{\Rightarrow} aabbcBd \underset{左}{\Rightarrow} aabbccdd$

(2)　$S \underset{左}{\Rightarrow} C \underset{左}{\Rightarrow} aCd \underset{左}{\Rightarrow} aaDdd \underset{左}{\Rightarrow} aabDcdd \underset{左}{\Rightarrow} aabbccdd$

これらに対応する構文木を，図 5.23 に示す．

L が本質的にあいまいであることの証明は複雑である．しかし要点は，以下の通りである．a, b, c, d がすべて同数の列は，有限個を除いて，二つの最左導出を持つ—— 一つの方法は a と b が同数になるように（A を使って）生成し，c と d が同数になるように（B を使って）生成する．もう一つの方法は，a と d が同数になるように（C を使って）生成し，b と c が同数になるように（D を使って）生成する．

たとえば a と b が同数になるように生成するには，図 5.22 の変数 A を使うしかない．もちろんいろいろな修正は可能であるが，どの修正にしても基本的な方針は変わらない．たとえば：

- いくつかの短い列は，基本的な生成規則を $A \to ab$ からたとえば $A \to aaabbb$ に変えることによって，省ける．
- 変数 A の仕事は，他の変数に共有させることができる．たとえば変数 A_1 と A_2 を使い，規則 $A_1 \to aA_2b \,|\, ab$; $A_2 \to aA_1b$ を導入すると，変数 A_1 は奇数個の a と b を生成し，A_2 は偶数個の a と b を生成するようにできる．
- A から生成される a と b の数が，正確に一致せず，ある有限の数だけ違っているようにすることもできる（前後で調整できる）．たとえば，生成規則 $S \to AbB$ から出発して，$A \to aAb \,|\, a$ によって，b より一つだけ多い a を生成するようにしてもよい．

しかし，a と同じ個数の b を生成する何らかの機構は，どうしても避けられない．

同様に，同数の c と d を生成する B のような変数が必要であることがいえる．さらに C の役割（a と同じ個数の d を生成する）をはたす変数も，D の役割（b と同じ個数の c を生成する）をはたす変数も必要である．これらの議論を精密にすると，どんな修正を施しても，$a^nb^nc^nd^n$ という形の列の少なくともいくつかは，図 5.22 の文法と同様に，2 通りの仕方で生成されることが示せる．

5.4.5　5.4 節の演習問題

♣ ***問 5.4.1**　文法

$$S \to aS \,|\, aSbS \,|\, \varepsilon$$

の文法は，あいまいである．列 aab が二つの

a)　構文木，

b)　最左導出，

c)　最右導出

を持つことを示せ.

♣ **! 問 5.4.2** 問題 5.4.1 の文法が，a と b の列で，先頭を含むどの一部分も a の数が b の数以上であるもののすべて（またそれらだけ）を生成することを証明せよ.

♣***! 問 5.4.3** 問題 5.4.1 の言語を生成するあいまいでない文法を設計せよ.

♣ **!! 問 5.4.4** 問題 5.4.1 の文法では，ある列は，ただ一つの構文木を持つ．与えられた列がその仲間であるかどうかを判定する，効率のよい判定法を示せ．なお,「すべての構文木を調べて，その成果のうち与えられた性質を満たすものを探す」方法は，十分効率的とはいえない.

♣ **! 問 5.4.5** 関係する問題 5.1.2 の文法を，再掲する：

$$S \rightarrow A1B$$

$$A \rightarrow 0A \mid \varepsilon$$

$$B \rightarrow 0B \mid 1B \mid \varepsilon$$

a) この文法があいまいでないことを示せ.

b) 同じ言語を生成するあいまい**である**文法を作り，それが事実あいまいであることを証明せよ.

♣***! 問 5.4.6** 問題 5.1.5 で作った文法はあいまいか？ もしあいまいであったら，それをあいまいでない文法に修正せよ.

♣ **問 5.4.7** 次の文法は，被演算項 x，y と 2 項演算 +，−，∗ の「演算前置き」記法の式を生成する：

$$E \rightarrow +EE \mid *EE \mid -EE \mid x \mid y$$

a) 列 **+*-xyxy** のための最左導出，最右導出，構文木を求めよ.

!b) この文法があいまいでないことを証明せよ.

5.5 第 5 章の要約

◆**文脈自由文法 CFG**：CFG は，生成規則と呼ばれる再帰的な規則によって言語を記述する手段である．CFG は変数の集合，終端記号の集合，出発記号，および生成規則の集合から成る．どの生成規則も，頭部の変数と 0 個以上の変数あるいは終端記号（両者が混在してもよい）の列から成る本体によって構成される.

◆**導出と言語**：出発記号から始めて，ある生成規則の頭部にある変数をその本体に置き換えることを繰り返して，終端記号列を導く．CFG の言語とは，そのようにして導くことができる終端記号列の集合のことで，文脈自由言語と呼ばれる.

◆**最左および最右導出**：いつでも列の最も左にある（あるいは最も右にある）変数を置き換えるとき，その導出を最左（あるいは最右）導出という．CFG の言語に属する列はどれも，少なくとも一つの最左導出と少なくとも一つの最右導出を持つ.

◆**文形式**：導出の途中結果は，変数と終端記号の列である．それらの列を文形式という．もし導出が最左導出（あるいは最右導出）なら，途中の列を最左（あるいは最右）文形式という．

◆**構文木**：構文木は導出の要点を示す木である．内部の節点のラベルは変数で，葉のラベルは終端記号か ε である．内部の各節点にはある生成規則が対応していて，その節点のラベルを頭部に持ち，その子のラベルを左から右に読むとちょうどその生成規則の本体に一致する．

◆**構文木と導出の等価性**：ある終端記号列がある文法の言語に属するのは，それがある構文木の成果になっているときで，またそのときに限る．だから最左導出，最右導出，および構文木の存在は，CFG の言語に属する列をぴったり定義する，等価な条件である．

◆**あいまいな文法**：CFG によっては，ある終端記号列に対して，二つ以上の構文木が存在する，あるいは同じことであるが，二つ以上の最左導出または二つ以上の最右導出が存在することがある．そのような文法はあいまいであるといわれる．

◆**あいまいさの除去**：多くの実用的な文法では，典型的なプログラミング言語においてプログラムの構造を記述する文法がそうであるが，同じ言語を生成するあいまいでない文法を構成することができる．残念なことに「あいまいでない文法」は，同じ言語に対する最も簡単な「あいまいな文法」よりも，ずっと複雑になることが多い．またある文脈自由言語で，ふつう非常に技巧的なものであるが，どの文法もあいまいであるものが存在する．

◆**パーサー**：文脈自由文法はコンパイラを含むプログラミング言語の処理系の構築に非常に重要な役割を果たしている．YACC のようなツールは，CFG を入力とし，「パーサー」というコンパイラの構成要素を出力するが，これはコンパイル中のプログラムの構造を分析・生成するものである．

◆**文書型定義 DTD**：情報をウェブ文書として共有するための，完成目前の XML 標準は，そのような文書の構造を記述するために，DTD と呼ばれる，文書の中に意味論的なタグを入れ子式に埋め込む記法を持っている．DTD は本質的に文脈自由文法で，その言語は関連する文書のクラスである．

5.6　第 5 章の参考文献

文脈自由文法は，自然言語を記述する手段として，Chomsky [4] によってはじめて導入された．同じアイデアはほんの少し遅れて，コンピュータ言語——J. W. Backus [2] によるフォートラン，P. Naur [7] によるアルゴルの記述に用いられた．そのため CFG は,「バッカス・ナウア形式の文法」と呼ばれることがある．

文法のあいまいさは，D. C. Cantor [3] と R. W. Floyd [5] にほとんど同時に取

り上げられた．本質的あいまいさは M. Gross［6］がはじめて論じた．

CFG のコンパイラへの応用については［1］を見よ．DTD は XML のための標準文書［8］で定義されている．

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

[2] J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," *Proc. Intl. Conf. on Information Processing* (1959), UNESCO, pp. 125-132.

[3] D. C. Cantor, "On the ambiguity problem of Backus systems," *J. ACM* **9**:4(1962), pp. 477-479.

[4] N. Chomsky, "Three models for the description of language," *IRE Trans. on Information Theory* **2**:3(1956), pp. 113-124.

[5] R. W. Floyd, "On ambiguity in phrase -structure languages," *Comm. ACM* **5**:10(1962), pp. 526-534.

[6] M. Gross, "Inherent ambiguity of minimal linear grammars," *Information and Control* **7**:3(1964), pp. 366-368.

[7] P. Naur et al., "Report on the algorithmic language ALGOL60," *Comm. ACM* **3**:5(1960), pp. 299-314, *Comm. ACM* **6**:1(1963), pp. 1-17.

[8] World-Wide-Web Consortium, http://www.w3.org/TR/REC-xml (1998).

邦訳文献

[1] 原田賢一訳：コンパイラ I, II（サイエンス社，1990）．

6 プッシュダウン・オートマトン

文脈自由言語に対して，それをちょうど定義する，ある型のオートマトンが存在する．そのオートマトンは「プッシュダウン・オートマトン」と呼ばれるが，正則言語を定義する一つの手段である「ε-動作を持つ非決定性・有限オートマトン」の拡張である．プッシュダウン・オートマトンは本質的に，ε-NFAにスタックをつけ加えたものである．スタックは，データ構造「スタック」のように，その上端でだけデータを読み，書き加え，取り出すことができる．

この章では，プッシュダウン・オートマトンを二つの異なる形式で定義する：一つは有限オートマトンのように，入力列を「受理状態に到達する」ことによって受理するもので，もう一つは状態とは無関係に「スタックが空になる」ことによって受理するものである．これらのどちらでも，受理される言語は文脈自由言語に正確に一致する；いいかえれば，文法をそれと等価なプッシュダウン・オートマトンに変換したり，またその逆もできる．我々はまた，プッシュダウン・オートマトンの重要な一部として，決定性プッシュダウン・オートマトンを考える．これは正則言語のすべてを受理するが，すべてのCFLを受理することはできない．このオートマトンは典型的なコンパイラのパーサーの機構によく似ているので，決定性プッシュダウン・オートマトンによって言語構造のどんなものが認識でき，どんなものが認識できないかを観察するのは，とても大事なことである．

6.1 プッシュダウン・オートマトンの定義

この節ではプッシュダウン・オートマトンをまず直観的に導入し，それから形式的な構成を示す．

6.1.1 直観的な導入

プッシュダウン・オートマトンは本質的に，ε-動作の許される非決定性有限オートマトンに，一つの新しい能力：スタックをつけ加えて，そこに「スタック記号列」を記録できるようにしたものである．スタックの存在は，有限オートマトンとは違って，プッシュダウン・オートマトンが情報をいくらでも「記憶」できることを意味する．しかし，無制限の情報に自由に接近できる汎用のコンピュータとは違って，プッシュダウン・オートマトンはスタックの中の情報には「先入れ後出し」方式でしか接近できない．

その結果，適切なコンピュータ・プログラムでなら認識できるが，プッシュダウン・オートマトンでは認識できない言語が存在する．実際，プッシュダウン・オートマトンでは文脈自由言語のすべてを認識できるが，それしか認識できない．多くの言語は，いくつかの正則でない言語も含めて，文脈自由であるが，7.2節で示すように，簡単に記述できるのに文脈自由ではない言語も存在する．文脈自由でない言語の例としては，同じ個数の0,1,2の列の集合 $\{0^n1^n2^n \mid n \geq 1\}$ がある．

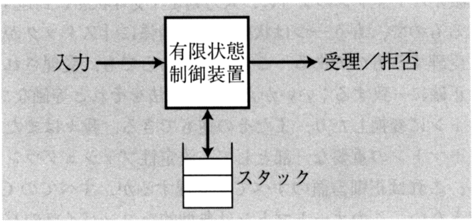


図 6.1　プッシュダウン・オートマトンとは本質的に，データ構造「スタック」を持った有限オートマトンである．

プッシュダウン・オートマトンは直観的には，図6.1が示すような装置と見ることができる.「有限状態制御装置」は，入力を1回に一つずつ読む．プッシュダウン・オートマトンはスタックの上端にある記号を読むことができ，またそのときの状態と入力記号，さらにスタックの上端の記号に基づいて，動作を決定することができる．その他，入力記号の代わりに空列 ε を入力とみなす「自発的な」動作を行うこともできる．1回の動作で，プッシュダウン・オートマト

ンは：

(1)　入力から一つの記号を消費して，以下の動作を決定する．ただし，入力として ε を使用する場合には，入力記号は消費されない．

(2)　新しい状態に移る：ただし前と同じ状態であってもさしつかえない．

(3)　スタックの上端の記号を，ある列に置き換える――その列には何の制限もない．もしその列が空列ならば，スタックからの「取り出し」（ポップアップ）が行われたことになる．その列がスタックの上端にあった記号と一致することもあるが，その場合はスタックは何も変わらない．上端の記号を他の一つの記号に置き換えることもある．その場合はスタックの上端の記号は変わるが,「取り出し」も「押し込み」（プッシュダウン）も起こらない．最後に，上端のスタック記号が 2 個以上の記号列に置き換えられた場合には，その結果は上端の記号を変え（もとのままもあり得る），さらにいくつかの新しい記号をスタックに押し込むことになる．

例 6.1　次の言語を考えよう．

$$L_{wwr} = \{ww^R \mid w \text{ は } (\mathbf{0}+\mathbf{1})^* \text{の中の列}\}$$

この言語は，字数が偶数の，アルファベット $\{0,1\}$ 上の回文（よく「w と w の反転」と呼ばれる）の集合である．これは CFL で，図 5.1 の文法から生成規則 $P \to 0$ と $P \to 1$ を省いた文法によって生成される．

この言語 L_{wwr} を受理するプッシュダウン・オートマトンを，設計してみよう[1]．

(1)　状態 q_0 から出発する．これは「回文の中央にまだ達していない」，すなわち「列 w を最後まで読んでいない」という "推測" を表す状態である．状態 q_0 では，入力記号を読み，そのコピーを順にスタックに押し込む――という形で入力を記録する．

(2)　いつでも,「中央まで達した」あるいは「w を読み終わった」と推測してよい．その時点で，w はスタックに，右端（最後）が上端，左端（先頭）が底に，記録されている．この推測は，状態を q_1 に移すことによって表

[1] 図 5.1 の文法で生成される言語 L_{pal} に対応するプッシュダウン・オートマトンを設計することもできる．しかし L_{wwr} のほうがいくらか単純で，プッシュダウン・オートマトンについての重要な概念に焦点をあてやすくなる．

現される．オートマトンは非決定性であるから，実際にはどちらの推測も可能である：w を終わりまで読んだ，と推測しながら，状態 q_0 にもとどまって，入力を読んでスタックに記録する作業も平行して続ける——と考えてもよい．

(3) 状態 q_1 に入ってからは，入力記号とスタックの上端の記号とを比較する．もしそれらが一致すれば，スタックの上端の記号を取り去って，次の入力記号に進む．また一致しない場合は，推測は正しくなかった：前に推測した w に続くのは，w^R ではなかった．この道筋はそれで終わるが，非決定性の他の道筋は生き残って，最終的には受理に成功することがあり得る．

(4) もしスタックが空になれば，入力 w のあとに w^R が続いていたのである．そこでそれまでに読み込んだ入力列を受理する．▨

6.1.2 プッシュダウン・オートマトンの形式的な定義

プッシュダウン・オートマトン（**PDA**）は形式的には，いくつかの構成要素から成る．我々は PDA P を次のように書き表す：

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

構成要素の意味は，次の通りである：

Q：**状態**の有限集合——これは有限オートマトンの場合と同じである．

Σ：**入力記号**の有限集合——これも有限オートマトンの入力記号に対応している．

Γ：有限の**スタック・アルファベット**．これは有限オートマトンにはないもので，スタックに記録できる記号の有限集合である．

δ：**推移関数**．有限オートマトンの場合と同じく，δ はオートマトンの動作を支配している．形式的には，δ は次のような三つの変数値に対する関数 $\delta(q, a, X)$ である：

(1) q は Q に属する状態である．

(2) a は Σ の中にある入力記号かまたは $a = \varepsilon$ である（空列 ε は入力記号ではないと仮定する）．

(3) X はスタック記号，すなわち Γ の要素である．

δ の出力は状態 p とスタック記号の列 γ の対 (p, γ) の集合で，p は

次の状態，γ はスタックの上端にある X を置き換える列を表している．たとえば，もし $\gamma=\varepsilon$ なら，スタックの上端 X は取り去られ（ポップされ），もし $\gamma=X$ ならスタックは変わらず，またもし $\gamma=YZ$ なら，X は Z に置き換えられ，その上に Y が押し込まれる（Z は Y の下になる）．

q_0：**初期状態**．PDA は何かの動作を行う前には，この状態にある．

Z_0：**開始記号**．PDA のスタックには，最初はこの記号 1 個だけがあって，他には何も入っていない．

F：**受理状態**（**最終状態**ともいう）の集合．

例 6.2　例 6.1 の言語 L_{wwr} を受理する PDA P を設計してみよう．スタックを適切に扱うためには，まず前の例では触れなかった細かい点がいくつか必要である．スタック記号 Z_0 は，スタックの底（下端）を明示するために利用できる．スタックから w を取り終わり，入力 ww^R を読んだと気づいた時点で，スタックの中に何かが残っていないと，受理状態 q_2 に推移することができない．そこで L_{wwr} に対する PDA は，次のようになる．

$$P=(\{q_0,q_1,q_2\},\ \{0,1\},\ \{0,1,Z_0\},\ \delta,\ q_0,\ Z_0,\ \{q_2\})$$

ここで δ は次の規則によって定義される：

(1)　$\delta(q_0,0,Z_0)=\{(q_0,0Z_0)\}$,　　$\delta(q_0,1,Z_0)=\{(q_0,1Z_0)\}$

状態が q_0 でスタックの上端に開始記号がみえているときは，これらの規則のどちらかがまず適用される．最初の入力記号を読みとり，開始記号をスタックの底に残したまま，その入力記号をスタックに押し込む．

(2)　$\delta(q_0,0,0)=\{(q_0,00)\}$,　　$\delta(q_0,0,1)=\{(q_0,01)\}$,

「まぜて組み合わせる」のはダメ

PDA は，状況によって，いくつかの選択肢を持つことができる．たとえば $\delta(q,a,X)=\{(p,YZ),\ (r,\varepsilon)\}$ としよう．PDA は一回の動作で，一つの対をまるごと選ばなければならない；状態を一つの対から，スタック記号の書き換えは他の対から選ぶことはできない．だから状態 q でスタックの上端に X があり，入力 a を読んだときは，状態 p に移って X を YZ に書き換えるか，さもなければ状態 r に移り X を消去することができる．しかし状態 p に移り X を消去することはできないし，状態 r に移って X を YZ に書き換えることも許されない．

$\delta(q_0,1,0)=\{(q_0,10)\}$,　$\delta(q_0,1,1)=\{(q_0,11)\}$

状態 q_0 では，入力記号を読み，それをスタックに（上端にある前の記号はそのままにして，その上から）押し込み，状態は変えない．

(3) $\delta(q_0,\varepsilon,Z_0)=\{(q_1,Z_0)\}$,　$\delta(q_0,\varepsilon,0)=\{(q_1,0)\}$,
$\delta(q_0,\varepsilon,1)=\{(q_1,1)\}$

これらの規則は，スタックの上端の記号が何であろうとそのままにして，P が状態 q_0 から状態 q_1 に自動的に（ε 入力で）推移することを許す．

(4) $\delta(q_1,0,0)=\{(q_1,\varepsilon)\}$,　$\delta(q_1,1,1)=\{(q_1,\varepsilon)\}$

状態 q_1 では，入力記号とスタックの上端の記号をつきあわせて，一致する場合は上端の記号を取り去る．

(5) $\delta(q_1,\varepsilon,Z_0)=\{(q_2,Z_0)\}$

最後に，P が状態 q_1 にあるときにスタックの底の目印が露わになったら，ww^R という形の入力列を読み込んだので，状態 q_2 に進んでそれを受理する． ▨

6.1.3 PDAの図表現

例 6.2 で示した δ 関係のリストは，そうわかりやすいものではない．有限オートマトンの推移図を一般化した図のほうが，与えられた PDA の振る舞いをもっと明瞭に示せることがある．そこで PDA のための**推移図**を導入し，すぐに使ってみよう．その図は：

a) PDA の状態に対応する節点，
b) 有限オートマトンと同様,「出発」と記された矢印は初期状態を表し，2 重の円は受理状態を表す．
c) PDA の状態推移に対応する弧は，次のように表される．$\delta(q,a,X)$ が対 (p,α) を（他の対と合わせて）含んでいるとき，q から p への弧にラベル a，X/α をつける．だから弧を見れば，どんな入力が使われ，スタックの上端記号がどのように変わるかがわかる．

図に示されないたった一つのことは，スタックの開始記号である．ふつう，特に断らない限り，開始記号は Z_0 である．

例 6.3 例 6.2 の PDA は，図 6.2 のような図で表現できる． ▨

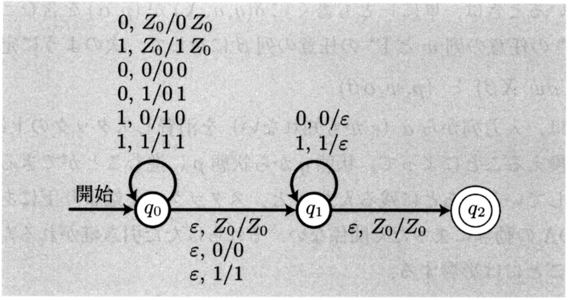


図 6.2 一般化された推移図による PDA の表現

6.1.4 PDA の時点表示

ここまでのところ PDA がどのように「計算するか」は，きちんと説明していない．直観的には，PDA は入力記号（ときには ε）に応じて状況を変えるが，どの状態にあるかさえわかればよい有限オートマトンとは違って，PDA の状況は状態とスタックの内容の両者に依存して決定される．いくらでも大きくなれる点で，スタックは PDA の状況全体の中でつねに，より重要な役割を果たすことが多い．状況を記述するとき，残された入力列を含めることも役に立つ．

そこで我々は PDA の状況を，次の三つの組 (q, w, γ) として定義する：

1. 状態 q,
2. 入力の残り w,
3. スタックの内容 γ

ふつう，γ の左端はスタックの上端を表し，右端がスタックの底を表す．このような三つの組はプッシュダウン・オートマトンの**時点表示**（instantaneous description，ID）と呼ばれる．

有限オートマトンの場合，その動きを表す時点表示の系列を表すには，有限オートマトンの時点表示はその状態だけなので，$\hat{\delta}$ 記法で十分であった．しかし PDA の場合，状態と入力，スタックの変化を記述する記法が必要になる．そこで「PDA の 1 回，あるいはそれ以上の動作」を表す「時点表示の対」をつなぐ「回転扉」記法を採用する．

$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ を PDA とする．$\vdash_P$ を次のように定義する（P が

わかっているときは，単に$\vdash$とも書く）：$\delta(q,a,X)$が(p,α)を含むとする．そのときΣ^*の任意の列wとΓ^*の任意の列βに対して，次のように定める：

$$(q,aw,X\beta)\ \vdash\ (p,w,\alpha\beta)$$

この動作は，入力列からa（εかも知れない）を消費しスタックの上端のXをαに置き換えることによって，状態qから状態pに進むことができる，ということを表している．あとに残る入力wと，スタックの上端より下にある記号列βは，PDAの動きにまったく関係ない．それらはただ引き継がれるだけで，あとのできごとには影響する．

PDAの0ステップ以上の動作を表すには，記号$\overset{*}{\underset{P}{\vdash}}$（$P$がわかっている場合は$\overset{*}{\vdash}$）を用いる．すなわち：

基礎：任意の時点表示Iに対して，$I\overset{*}{\vdash}I$

帰納：ある時点表示I,J,Kについてもし$I\vdash K$でしかも$K\overset{*}{\vdash}J$ならば，$I\overset{*}{\vdash}J$

いいかえればもしある時点表示の列$K_1,K_2,\cdots,K_n$について$I=K_1$，$J=K_n$で，しかも$j=1,2,\cdots,n-1$について$K_j\vdash K_{j+1}$ならば，$I\overset{*}{\vdash}J$である．

例 6.4 例6.2のPDAの，入力1111に対する動き方を考えてみよう．q_0が初期状態，Z_0が開始記号であるから，最初の時点表示は$(q_0,1111,Z_0)$である．この入力に対して，PDAは間違った推測を何回も行う可能性がある．最初の時点表示$(q_0,1111,Z_0)$から到達し得るすべての時点表示の系列は，図6.3に示した．矢印は関係$\vdash$を表している．

最初の時点表示から，二つの動作を選択できる．第1の動作（図の左側）では，「入力の中央にはまだ達していない」と推測して，時点表示$(q_0,111,1Z_0)$に進む．その結果，1個の1が入力から取り去られ，スタックの上に押し込まれた．

最初の時点表示からの第2の選択（ナナメ右下に進む矢印）では，「中央に達した」と推測する．入力を消費せずに，は状態q_1に進み，時点表示は$(q_1,1111,Z_0)$となる．この状態でスタックの上端にZ_0があればPDAは「受理する」ので，PDAはそこから時点表示$(q_2,1111,Z_0)$に進む．しかし入力がまだ残っているので，この時点表示は正しい「受理・時点表示」ではない．入力が1111ではなくてもしεであれば，この同じ動きで時点表示(q_2,ε,Z_0)に到達し，受理されたとわかるはずである．

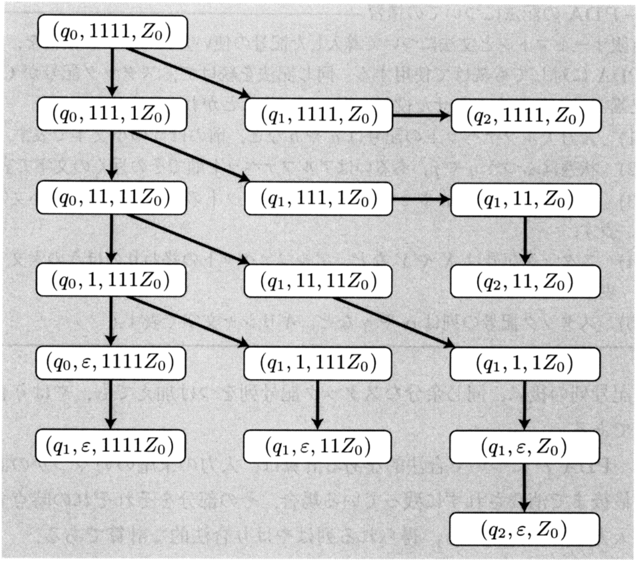


図 6.3 例 6.2 の PDA の，入力 1111 に対する時点表示

PDA は，1 個の 1 を読んで時点表示 $(q_0, 111, 1Z_0)$ に進んでから「中央に到達した」と推測することもできる．この推測も，すべての入力を読まないので，失敗する．2 個の 1 を読んだときに「中央に到達した」と正しく推測すれば，次のような時点表示の列が得られる．

$$
\begin{aligned}
&(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \\
\vdash\ &(q_1, 11, 11Z_0) \vdash (q_1, 1, 1Z_0) \vdash (q_1, \varepsilon, Z_0) \\
\vdash\ &(q_2, \varepsilon, Z_0) \quad ▨
\end{aligned}
$$

PDA について議論をするために重要な，時点表示とそれらの推移について三つの原則がある．

(1) PDA P について合法的な時点表示の列（**計算**）は，それぞれの時点表示の入力の後に，同じ余分な入力列をつけ加えても，やはり合法的である．

(2) PDA P について合法的なある計算は，それぞれの時点表示のスタック

PDA の記法についての慣習

有限オートマトンと文法について導入した記号の使い方についての慣習を，我々は PDA に対しても続けて使用する．同じ記法を続けて，スタック記号が CFG の終端記号と変数をあわせた役割を担っていることがわかるとよい．

(1)　入力アルファベットの記号は a や b など，前のほうの小文字で表す．

(2)　状態はふつう q や p，あるいはアルファベット順でその近くの文字で表す．

(3)　入力記号の列は w や z など，アルファベットの終わりのほうの小文字で表す．

(4)　スタック記号は X や Y など，アルファベットの終わりのほうの大文字で表す．

(5)　スタック記号の列は α や γ など，ギリシャ文字で表す．

記号列の後に，同じ余分なスタック記号列をつけ加えても，やはり合法的である．

(3)　PDA P について合法的なある計算は，入力の末尾のいくつかの記号が最後まで消費されずに残っている場合，その部分をそれぞれの時点表示の入力から削除しても，得られる列はやはり合法的な計算である．

直観的にいうと,「P が見ていない情報は，計算に影響しない」ということである．これらのうち (1) と (2) は，一つの定理にまとめておいた．

定理 6.5　$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ を PDA とする．もし

$$(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$$

ならば，Σ^* の中の任意の w と Γ^* の中の任意の γ に対して

$$(q, xw, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta\gamma)$$

$\gamma = \varepsilon$ の場合が前に述べた原則 (1) で，$w = \varepsilon$ の場合が原則 (2) である．

証明　証明は，$(q, xw, \alpha\gamma)$ を $(p, yw, \beta\gamma)$ に移す時点表示の列のステップ数についての，ごく簡単な帰納法でできる．列 $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$ のどのステップでも，P の推移は w にまったく関係しない．だからそれらをつけ加えた $(q, xw, \alpha\gamma)$ から $(p, yw, \beta\gamma)$ への推移としても，正当化される．▨

ついでながらこの定理の逆は，γ については正しくないことを注意しておこう．PDA が γ の一部を取り出し，書き換えることによってできることで，γ なしにはできないことがあり得る．しかし入力については「消費した記号をまた

有限オートマトンに時点表示はないのか？

我々が使っている PDA の時点表示を，有限オートマトンに対して導入しなかったのはどうしてなのか，不思議に思う人もいるかもしれない．有限オートマトンはスタックを持っていないが，状態 q と残りの入力 w の対 (q, w) を有限オートマトンの時点表示として使うこともできた．

そのようにしても，有限オートマトンの到達可能性について，記法 $\hat{\delta}$ から得られる情報よりも多くのことが明らかになるわけではない．つまり，すべての列 x について $\hat{\delta}(q, w) = p$ であることと $(q, wx) \overset{*}{\vdash} (p, x)$ であることはまったく同等なのである．「x が何であっても有限オートマトンの動作には影響しない」という事実は，定理 6.5，6.6 で述べた事実に対応する定理である．

戻す」ことはあり得ないので，原則 (3) がいっているように，使わなかった入力を削除することができる．これは形式的には，次のように書ける：

定理 6.6 $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ を PDA とする．もし

$$(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$$

ならば，

$$(q, x, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, y, \beta\gamma)$$

6.1.5 6.1 節の演習問題

♣ **問 6.1.1** PDA $P = (\{q, p\}, \{0, 1\}, \{Z_0, X\}, \delta, q, Z_0, \{p\})$ が，次のような推移関数を持っているとする：

1. $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$
2. $\delta(q, 0, X) = \{(q, XX)\}$
3. $\delta(q, 1, X) = \{(q, X)\}$
4. $\delta(q, \varepsilon, X) = \{(p, \varepsilon)\}$
5. $\delta(p, \varepsilon, X) = \{(p, \varepsilon)\}$
6. $\delta(p, 1, X) = \{(p, XX)\}$
7. $\delta(p, 1, Z_0) = \{(p, \varepsilon)\}$

最初の時点表示 (q, w, Z_0) から出発して，次の入力 w について，どのような時点表示に到達できるかを示せ．

*a) 01

b) 0011

c) 010

6.2 PDA の言語

PDA が入力を受理するのは，その入力を最後まで読みとって受理状態に入ったときである．この方式は「最終状態による受理」と呼ばれる．PDA の言語を定義するもう一つの方式があって，重要な応用がある．それは PDA が「空スタックによって受理する」言語，すなわち,「初期時点表示から出発して，スタックを空にするような入力列」の集合を考える方式である．

これら二つの方式は，ある PDA によって「最終状態で受理」される言語は，別の PDA によって「空スタックで受理」されるし，その逆も成り立つという意味で，同等である．しかし，一つの PDA P が「最終状態で受理」する言語と,「空スタックで受理」する言語とは，一般に異なる．この節では，言語 L を「最終状態で受理」する PDA から，同じ言語を「空スタックで受理」する PDA を構成する方法を示し，またその逆を行う方法を示す．

6.2.1 最終状態による受理

$P=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ を PDA とする．これが**最終状態で受理する言語** $L(P)$ とは，

$$\{w \mid F \text{ に属するある状態 } q \text{ について } (q_0,w,Z_0) \overset{*}{\underset{P}{\vdash}} (q,\varepsilon,\alpha)\}$$

のことである．ただし α は任意のスタック記号列である．だから受理の条件は，w が入力として待ち受けている最初の時点表示から出発して，P が w を消費し尽くしてある受理状態になることである．そのときのスタックの内容は関係ない．

例 6.7　例 6.2 の PDA は言語 L_{wwr}，すなわち ww^R という形の $\{0,1\}$ 上の言語を受理する，と述べた．次に，これが正しいことを証明しよう．証明の目標は：例 6.2 の PDA が列 x を受理する必要十分条件は，x が ww^R という形をしていることである．

（十分性）この部分はやさしい：P が x を受理する計算過程を示せばよい．

$$(q_0,ww^R,Z_0) \overset{*}{\vdash} (q_0,w^R,w^RZ_0) \vdash (q_1,w^R,w^RZ_0)$$
$$\overset{*}{\vdash} (q_1,\varepsilon,Z_0) \vdash (q_2,\varepsilon,Z_0)$$

このように，PDA の一つの可能な道筋は，まず入力から w を読んでスタックに逆順に記録する（上から押し込むので，自然に逆順になる）．次に自動的に

状態 q_1 に移り，入力の w^R とスタックの中の同じ列をつき合わせ，最後に（スタックが Z_0 だけになったところで）自動的に状態 q_2 に移る．

（必要性）この部分のほうがむずかしい．まず，状態 q_2 に移れるのは「状態 q_1 でスタックの上端が Z_0 であるときに限る」ことを注意しておこう．また P は x を受理するまでに，状態 q_0 から出発した途中で一度 q_1 に移り，そのあと q_0 にはけっして戻らない．だから，列 x が $(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \varepsilon, Z_0)$ を満たす条件をみつければよい：P が最終状態で受理するのはこのような列 x である．我々はこれより少し一般的な主張を，x の長さについての帰納法で証明する．

- もし $(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \alpha)$ ならば，x は ww^R という形の列である．

基礎：もし $x = \varepsilon$ ならば，x は $w = \varepsilon$ について，ww^R という形になっている．だからこの結論部分は正しく，主張は正しい．なお前提 $(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \alpha)$ が正しいかどうかは（この場合，事実正しいが）吟味するまでもない．

帰納：ある $n > 0$ について $x = a_1 a_2 \cdots a_n$ と仮定する．時点表示 (q_0, x, α) からは，次の二つの動きが可能である．

(1) $(q_0, x, \alpha) \vdash (q_1, x, \alpha)$． 状態 q_1 では，P はスタックから記号を取り去るので，x の長さが 1 以上であるから，$(q_1, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \beta)$ ならば β は α より短く，α と等しくはならない．

(2) $(q_0, a_1 a_2 \cdots a_n, \alpha) \vdash (q_0, a_2 \cdots a_n, a_1\alpha)$．今度は，最終的に $(q_1, \varepsilon, \alpha)$ に到達するためには，その直前のステップは

$$(q_1, a_n, a_1\alpha) \vdash (q_1, \varepsilon, \alpha)$$

しかあり得ない．その場合は $a_n = a_1$ で，しかも

$$(q_0, a_2 \cdots a_n, a_1\alpha) \vdash (q_1, a_n, a_1\alpha)$$

でなければならない．定理 6.6 から，使われなかった入力の末尾は削ってよいので，

$$(q_0, a_2 \cdots a_{n-1}, a_1\alpha) \vdash (q_1, \varepsilon, a_1\alpha)$$

この推移での入力の長さは n より小さいので，帰納法の仮定から $a_2 \cdots a_{n-1}$ はある y について yy^R という形をしている．$x = a_1 yy^R a_n$ でしかも $a_n = a_1$ であるから，x は ww^R と表せる（$w = a_1 y$ とおけばよい）．

以上が「x が受理されるのは，x がある w について $x = ww^R$ と表せるときに限る」ことの証明の核心部分である．このように，必要性が証明でき，十分性

はすでに証明されているので，P がちょうど言語 L_{wwr} の中の列を受理することが証明できた．▨

6.2.2 空スタックによる受理

PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ に対して，**状態空スタックで受理する言語** $N(P)$ を次のように定義する[2]：

$$\{w \mid F \text{ に属するある状態 } q \text{ について } (q_0, w, Z_0) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)\}$$

すなわち $N(P)$ は「P がそれを消費しつくして，しかもスタックを空にできるような入力列 w」の集合である．

例 6.8　例 6.2 の PDA P は，スタックが空にならない．だから $N(P) = \varnothing$ である．しかしわずかの修正で，P が最終状態でも空スタックでも L_{wwr} を受理するようにできる．それには推移 $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$ を $\delta(q_1, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}$ に変更すればよい．そうすれば，最後にスタック記号が取り去られるので，

$$L(P) = N(P) = L_{wwr}$$

となる．▨

空スタックによる受理だけを考える場合には，最終状態は関係ないので，PDA P の定義の最後の（第7の）要素 F を省略することがある．だから P は六つの組 $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ で表される．

6.2.3 空スタックから最終状態へ

ある PDA P によって $L(P)$ で表される言語のクラスが，適当な PDA P によって $N(P)$ で表される言語のクラスと一致することを示そう．このクラスは，6.3 節で示すが，文脈自由言語のクラスとも一致する．我々はまず，ある言語 L を空スタックで受理する PDA P_N が与えられたとして，そこから同じ言語 L を最終状態で受理する PDA P_F を構成する方法を示す．

定理 6.9　もし PDA $P_N = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ について $L = N(P_N)$ であるなら，$L = L(P_F)$ を満たす PDA P_F が存在する．

証明　証明の背後にあるアイデアを図 6.4 に示した．ここで X_0 は，Γ の中にない，新

---

[2] $N(P)$ の N は「空スタック」（empty stack）の同義語 "null stack" の頭文字である．

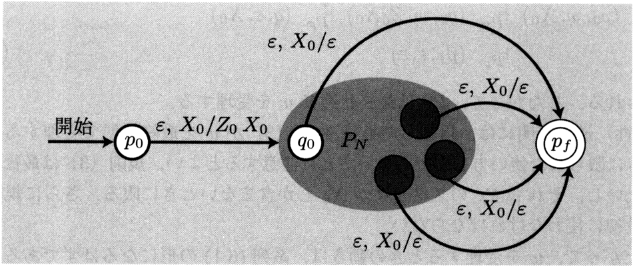


図 6.4 P_N を模倣し，P_N のスタックが空になったとき入力列を受理する P_F

しい記号である．X_0 は P_F の出発記号であり，P_N が空スタックに到達したことを知らせる「底の目印」でもある．すなわち，もし P_F がスタックの上端に X_0 を見たら，それは P_N が同じ入力に対してスタックを空にすることを意味している．

新しい出発状態 p_0 も必要である：その役目は，P_N の出発記号 Z_0 をスタックに押し込み，P_N の初期状態 q_0 に移ることだけである．P_F は P_N のスタックが空になるまで P_N のシミュレーションを行い，スタックが空になったことを上端に X_0 が現れることによって認識し，それから新しい状態（の受理状態）p_f に移る：この p_f への推移は，P_N のスタックが空になったとわかればすぐに行う．

P_F の内容は，次の通りである．

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

ここで δ_F は次のように定義される．

(1) $\delta_F(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$
　P_F は，初期状態 p_0 から自動的に P_N の初期状態 q_0 に移り，開始記号 Z_0 をスタックの上に押し込む．

(2) Q の中の状態 q，Σ の中の記号 a または $a = \varepsilon$，および Γ の中のスタック記号 Y に対して，$\delta_F(q, a, Y)$ は $\delta_N(q, a, Y)$ の中のすべての対を含む．

(3) 規則（2）に加えて，Q の中の状態 q に対して，$\delta_F(q, \varepsilon, X_0)$ は (p_f, ε) を含む．

そこで次のことを証明しよう：w が $L(P_F)$ に属する必要十分条件は，w が $N(P_N)$ に属することである．

（十分性）ある状態 q について $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \varepsilon)$ と仮定する．定理 6.5 によれば，スタックの底に X_0 をつけ加えて，$(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, X_0)$ と考えてよい．上の規則（2）によれば，P_F は P_N のすべての動きができるので，

$$(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, X_0)$$

も成り立つ．これに規則（1）と（3）に示されている最初と最後の動きをつけ加えれば，

$$(p_0, w, X_0) \vdash_{P_F} (q_0, w, Z_0X_0) \overset{*}{\vdash}_{P_F} (q, \varepsilon, X_0) \vdash_{P_F} (p_f, \varepsilon, \varepsilon) \qquad (6.1)$$

が得られる．したがって，P_F は最終状態で w を受理する．

(必要性) 逆の証明には，追加した推移 (1)，(3) が w を最終状態で受理するためには非常に限られた使い方しかできないことに注意するとよい．規則 (3) は最後にしか使えないし，それも P_F のスタックが X_0 しか含まないときに限る．さらに規則 (1) は，最初に使わなければならない．

したがって，w を受理するときの動きは，系列 (6.1) の形になるはずである．さらに，最初と最後を除く途中の計算は，スタックの底に X_0 をおいた P_N の計算でもある．その理由は，最初と最後のステップを除いて，P_F のどんな推移も P_N の推移と同じであり，底にある計算が次のステップで終わるところを除いて，スタックの上端に現れることはないからである．したがって $(q_0, w, Z_0) \overset{*}{\vdash}_{P_N} (q, \varepsilon, \varepsilon)$ が成り立ち，w は $N(P_N)$ に属する．▨

例 6.10　C プログラムの `if` と `else` の列を処理する PDA を設計しよう．5.3.1 項でやったように，i で `if` を表し，e で `else` を表すことにする．`else` はそれより前の `if` と組を作るので，プログラムをどこで切っても，その中の `else` の数は `if` の数を越えない．だからそれまでに見た i と e の個数の差を，スタック記号 Z で数えることにしよう．この単純な 1 状態の PDA は，図 6.5 の推移図で示される．

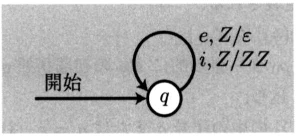


図 6.5　`if/else` の誤りを空スタックで受理する PDA

この PDA は，i を読むとスタックの Z を一つ増やし，e を読むと Z を一つ消す．最初はスタックに 1 個だけ Z があるとすれば，スタックの Z が n 個になるのは i の個数が e より $n-1$ 個だけ多かったときである．特に，スタックが空になるのは e のほうが i より一つだけ多かったとき，つまりそれまでに読んだ入力が最初に規則違反になったときである．この PDA が空スタックで受理するのはそのような列である．この P_N を形式的に述べると：

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

ここで δ_N は次のように定義される．

1. $\delta_N(q, i, Z) = \{(q, ZZ)\}$ i を読んだら Z を一つ押し込む．
2. $\delta_N(q, e, Z) = \{(q, \varepsilon)\}$ e を読んだら Z を一つ消す．

次に，この PDA P_N から，同じ言語を最終状態で受理する PDA P_F を構成してみよう．P_F の推移図は図 6.6 に示す通りである[3]．ここでは新しい初期状態 p，受理状態 r を導入している．またスタックの底の目印として X_0 を使っている．P_F は形式的には次のように述べられる：

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

ここで δ_F は次のように定義される．

1. $\delta_F(p, \varepsilon, X_0) = \{(q, ZX_0)\}$ この規則で，P_F による P_N のシミュレーションが開始される．
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$ P_N の模倣：i を読むと Z を押し込む．
3. $\delta_F(q, e, Z) = \{(q, \varepsilon)\}$ P_N の模倣：e を読むと Z を消す．
4. $\delta_F(q, \varepsilon, X_0) = \{(r, \varepsilon)\}$ もとの P_F のスタックが空になったとき，P_F は受理する． ▨

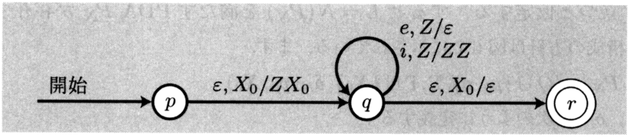


図 6.6 図 6.5 の PDA から作った最終状態で受理する PDA

6.2.4 最終状態から空スタックへ

今度は，逆方向を考える：ある言語 L を最終状態で受理する PDA P_F が与えられたとして，同じ言語 L を空スタックで受理する PDA P_N を構成したい．それは簡単で，図 6.7 で示した．P_F の受理状態から，ε-動作で新しい状態 p に移る．状態 p では，P_N は入力を消費せずにスタックの中を空にする．だから

---

[3] 定理 6.9 では新しい状態として p_0 と p_f を使っているのに，ここでは p と r を使っているが，気にしてはいけない：状態の名前など，もちろん何でもいいのである．

P_F が入力列 w を読み終わってから受理状態に入ったとき，P_N は w を読み終わってからスタックを空にする．

P_F が受理状態に入らないで偶然にスタックを空にしたとき，P_N がそれを真似てスタックを空にしないように，P_N もスタックの底に目印 X_0 をおく．この目印は P_N の開始記号で，定理 6.9 での構成と同様に，P_N は新しい状態 p_0 から出発し，まず P_F の開始記号をスタックに押し込んで P_F の初期状態に移る．構成の概略は図 6.7 に示すが，形式的には次の定理で述べる．

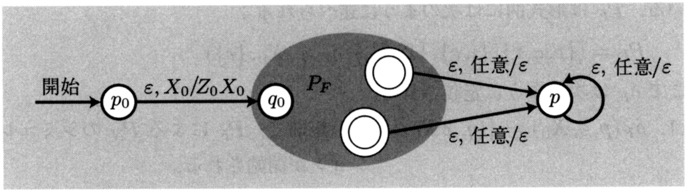


図 6.7 P_F のシミュレーションを行い，P_F が受理状態に入ったときにスタックを空にする P_N

定理 6.11 ある PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$ について，$L = L(P_F)$ が成り立つと仮定する．すると $L = N(P_N)$ を満たす PDA P_N が存在する．

証明 構成の方針は図 6.7 に示されている．まず

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

と置き，δ_N を次のように定義する：

(1) $\delta_N(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$．最初は P_F の開始記号をスタックに押し込み，P_F の初期状態に移る．

(2) Q の中の状態 q，Σ の入力記号 a あるいは $a = \varepsilon$，また Γ の中の記号 Y に対して，$\delta_N(q, a, Y)$ は $\delta_F(q, a, Y)$ の中のすべての対を含む．すなわち，P_N は P_F を模倣する．

(3) F のすべての受理状態 q とスタック記号 Y，あるいは $Y = X_0$ について，$\delta_N(q, \varepsilon, Y)$ は (p, ε) を含む．この規則によって，P_F が受理するときはいつでも，P_N は入力を読まずにスタックを空にする作業を始めることができる．

(4) Γ の中のすべてのスタック記号 Y，あるいは $Y = X_0$ について，$\delta_N(p, \varepsilon, Y) = \{(p, \varepsilon)\}$．状態 p に入るのは，P_F が受理するときに限る．そこでこの状態に入ると，P_N はスタックの記号をすべて消去する．入力は何も消費しない．

次に,「w が $N(P_N)$ に属する必要十分条件は w が $L(P_F)$ に属することである」ことを証明しなければならない．方針は定理 6.9 の場合と同様である.「十分性」は直接のシミュレーションで示せるし,「必要性」は構成された PDA P_N ができることのうちいくつか限られたことを吟味すればよい．

（十分性）ある受理状態 q とスタック記号列 α について $(q_0, w, Z_0)) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, \alpha)$ と仮定する．P_N は P_F のすべての動きを模倣できるので，定理 6.5 を援用してスタックの底の X_0 はそのままにして，$(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0)$ が成り立つ．だから次のことも成り立つ：

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \varepsilon, \varepsilon)$$

最初の動作はを構成する規則（1）で，最後の動きは規則（3）と（4）による．このように，P_N は空スタックで w を受理する．

（必要性）スタックの底には，もともとの Γ にない記号 X_0 があって，P_F はこれを変えられない．だからスタックが空になるのは，P_N が状態 p に推移してからである．P_N が状態 p に推移できるのは，シミュレーションの相手の P_F が状態 p に推移が受理状態に達したときしかない．P_N の最初の動作は規則（1）で規定されている動作なので，P_N の受理に至る計算は次の形になる：

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \varepsilon, \varepsilon)$$

ここで q は P_F の受理状態である．

さらに，時点表示 (q_0, w, Z_0X_0) から $(q, \varepsilon, \alpha X_0)$ までの動きは，すべての動作は P_F の動作である．特に，記号 X_0 は時点表示 $(q, \varepsilon, \alpha X_0)$ に到達するまでの間，スタックの上端にはけっして現れない[4]．だから，同じ計算——すなわち $(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, \alpha X_0)$ が P_F でも起こると結論してよい．このように P_F は最終状態で w を受理する．だから w は $L(P_F)$ に属している． ▨

6.2.5 6.2 節の演習問題

♣ **問 6.2.1** 次の言語を受理する PDA を設計せよ．最終状態でも空スタックでも，やりやすいほうを使ってかまわない．

- *a) $\{0^n1^n \mid n \geq 1\}$.
- b) 0 と 1 の列で，先頭からどの位置までを見ても，1 の個数が 0 の個数を超えないものの集合．
- c) 同じ個数の 0 と 1 を含む列の集合．

♣ **! 問 6.2.2** 次の言語を受理する PDA を設計せよ．

- *a) $\{a^ib^jc^k \mid i = j$ または $j = k\}$. 問 5.1.1（b）とは異なるので注意．
- b) 0 の数が 1 の数のちょうど 2 倍であるような列の集合．

---

4) P_F が受理するときにちょうどスタックが空であれば，α は ε である．

♣!! **問 6.2.3**　次の言語を受理する PDA を設計せよ．

a)　$\{a^i b^j c^k \mid i \neq j$ または $j \neq k\}$.

b)　a と b の列で，ww の形をしていないもの，言い換えればどんな列の繰り返しにも一致しないような列の集合．

♣*! **問 6.2.4**　PDA P がある空列を含まない言語 L を空スタックで受理すると仮定する：$L = N(P)$．この P を修正して，$L \cup \{\varepsilon\}$ を空スタックで受理させるには，どうすればよいか．方法を述べよ．

♣　**問 6.2.5**　PDA $P = (\{q_0, q_1, q_2, q_3, f\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$ の δ が，次の規則で定められるとする：

$$\begin{array}{ll}
\delta(q_0, a, Z_0) = (q_1, AAZ_0), & \delta(q_0, b, Z_0) = (q_2, BZ_0), \\
 & \delta(q_0, \varepsilon, Z_0) = (f, \varepsilon); \\
\delta(q_1, a, A) = (q_1, AAA), & \delta(q_1, b, A) = (q_1, \varepsilon), \\
 & \delta(q_1, \varepsilon, Z_0) = (q_0, Z_0); \\
\delta(q_2, a, B) = (q_3, \varepsilon), & \delta(q_2, b, B) = (q_2, BB), \\
 & \delta(q_2, \varepsilon, Z_0) = (q_0, Z_0); \\
\delta(q_3, \varepsilon, B) = (q_2, \varepsilon), & \delta(q_3, \varepsilon, Z_0) = (q_1, AZ_0);
\end{array}$$

どの場合も動作の選択肢は一つしかないので（右辺の集合の要素がどれも一つ），右辺の集合の記号 $\{\,\}$ を省略した．

*a)　bab が $L(P)$ に属することを示す，実行の経過（時点表示の列）を書け．

b)　abb が $L(P)$ に属することを示す，実行の経過を書け．

c)　P が入力 b^7a^4 を読んだあとの，スタックの内容を書け．

!d)　$L(P)$ がどんな集合か，直観的に説明せよ．

♣　**問 6.2.6**　問 6.1.1 の PDA P について考える．

a)　P が最終状態で受理するのと同じ言語を，空スタックで受理する PDA P_1（つまり $N(P_1) = L(P)$ となる P_1）を作れ．

b)　$L(P_2) = N(P)$ を満たす P_2（つまり P が空スタックで受理する言語を最終状態で受理する P_2）を作れ．

♣ ! **問 6.2.7**　P が PDA ならば，P と同じ言語を受理する PDA P_2（つまり $L(P_2) = L(P)$）でスタック記号が二つしかないものが存在することを示せ．**ヒント**：スタック記号の 2 進符号化．

♣*! **問 6.2.8**　**制限つき** PDA とは，どんな推移でも，スタックの高さがせいぜい 1 記号しか増えない，いい換えれば，任意の規則 $\delta(q, a, Z)$ に含まれる対 (p, γ) が $|\,\gamma\,| \ \leq 2$ を満たすような PDA のことである．任意の PDA P に対して，$L(P_3) = L(P)$ を満たす制限つき PDA P_3 が存在することを示せ．

6.3 PDA と CFG の等価性

次に，PDA によって定義される言語がちょうど文脈自由言語になることを示そう．攻撃の方針は図 6.8 に示した．証明の目標は，次の三つの言語のクラスがすべて一致することである：

(1) 文脈自由言語，すなわち CFG で定義される言語，

(2) ある PDA によって，最終状態で受理される言語，

(3) ある PDA によって，空スタックで受理される言語．

(2) と (3) が同じであることはすでに証明されている．次は (1) と (3) が同じであることを示すのが一番やさしい——それで三つのクラスがすべて等しいことの証明になる．

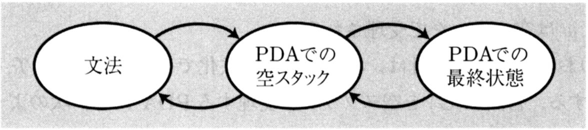


図 6.8 CFL を定義する三つの方法の等価性を示す構成の構造

6.3.1 文法からプッシュダウン・オートマトンへ

ある CFG G が与えられたとして，その G の最左導出を模倣する PDA を構成しよう．終端記号列ではない任意の左文形式は，$xA\alpha$ という形に書ける：ここで A は最も左に現れる変数，x はそれより左にある終端記号の列，α は A より右にある終端記号と変数の列である．$A\alpha$ をこの左文形式の**尾**と呼ぶ．もし左文形式が終端記号列なら，尾は空列である．

文法から PDA を構成する基本的なアイデアは，終端記号列 w を生成するのに文法が使う左文形式の列を，PDA に模倣させることである．各文形式 $xA\alpha$ の尾 $A\alpha$ は，スタックの中に A を先頭（左端）として記録される．その時点で，x はすでに消費された入力の一部（終端記号列）を「表現」している．あとに続く入力は，w の前方の x を削った残りである．だから，もし $w = xy$ ならば，y が残りの入力である．

PDA が左文形式 $xA\alpha$ に対応する時点表示 $(q, y, A\alpha)$ にあるときを考えてみよう．PDA は A を展開する生成規則，たとえば $A \to \beta$ の模倣を予測すると，

スタックの上端の A を β に書き換え，時点表示 $(q, y, \beta\alpha)$ に移る．なおこのPDA の状態は，一つしかない．

さて $(q, y, \beta\alpha)$ は，β の左端が変数とは限らないので，必ずしも左文形式を表現していない．実際，β が変数を全く含まない場合もあり得るし，α の左端も終端記号かも知れない．スタックの上端に変数が現れるように，$\beta\alpha$ の先頭部分にある終端記号を除去しなければならない．そこでそれらの終端記号を入力と照合して，入力列 w の最左導出として予測が正しいかどうかを確認する．もし照合に失敗したら，PDA のその道筋はそこで途絶える．

もしこのように w の最左導出を予測するのに成功したら，最終的に左文形式 w に到達する．その時点で，スタックのすべての記号は（変数なら）展開され（終端記号なら）入力と照合して，すべて消去されている．つまりスタックは空なので，w は空スタックで受理される．

以上のおおざっぱな方針は，次のように定式化できる．$G = (V, T, Q, S)$ をCFG とする．言語 $L(G)$ を空スタックで受理する PDA P を，次のように構成する．

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

ここで推移関数 δ は次のように定義する：

(1)　各変数 A について，

$$\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \rightarrow \beta \text{が } P \text{ の生成規則 }\}$$

(2)　各終端記号 a について，

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

例 6.12　図 5.2 の文法を PDA に変換してみよう．文法は：

$$I \;\rightarrow\; a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

$$E \;\rightarrow\; I \mid E * E \mid E + E \mid (E)$$

PDA の終端記号の集合は $\{a, b, 0, 1, (,), +, *\}$ である．これらの八つの記号と記号 I，E がスタック記号となる．PDA の推移関数は：

a)　$\delta(q, \varepsilon, I) = \{(q, a),\ (q, b),\ (q, Ia),\ (q, Ib),\ (q, I0),\ (q, I1)\}$.

b)　$(q, \varepsilon, E) = \{(q, I),\ (q, E + E),\ (q, E * E),\ (q, (E))\}$.

c)　$\delta(q, a, a) = \{(q, \varepsilon)\}$，$\delta(q, b, b) = \{(q, \varepsilon)\}$，

$$\delta(q, 0, 0) = \{(q, \varepsilon)\},\ \ \delta(q, 1, 1) = \{(q, \varepsilon)\},$$

$$\delta(q, (, () = \{(q, \varepsilon)\},\ \ \delta(q,),)) = \{(q, \varepsilon)\},$$

$$\delta(q, +, +) = \{(q, \varepsilon)\},\ \ \delta(q, *, *) = \{(q, \varepsilon)\}.$$

(a) と (b) は規則 (1) に従って，また (c) の八つの推移規則は規則 (2) にしたがって作られた. (a) から (c) までに明記されていない場合は，δ は空集合である. ▨

定理 6.13 CFG G から上のようにして構成された PDA P は，$N(P) = L(G)$ を満たす.

証明 w が $N(P)$ に属する必要十分条件は，w が $L(G)$ に属することである．いつものように場合を分けて証明しよう.

（十分性） w が $L(G)$ に属すると仮定しよう．すると w を導く最左導出

$$S = \gamma_1 \underset{\text{左}}{\Rightarrow} \gamma_2 \underset{\text{左}}{\Rightarrow} \cdots \underset{\text{左}}{\Rightarrow} \gamma_n = w$$

がある．以下，i についての数学的帰納法で，次のことを証明する：

$x_i y_i = w$, $\gamma_i = x_i \alpha_i$を満たす y_iとα_iについて，

$$(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$$

が成り立つことを証明する（α_i は γ_i の尾で，y_i は x_i が消費された後の入力の残りである）.

基礎：$i = 1$ のときは $\gamma_1 = S$，したがって $x_i = \varepsilon$，$y_i = w$，$\alpha_1 = S$ である．0 回の動作で $(q, w, S) \vdash (q, w, S)$ となるので，基礎は正しい.

帰納：次に 2 番目以下の左文形式を調べよう.

$$(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$$

を仮定して，$(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_{i+1}, \alpha_{i+1})$ を証明したい．α_i は尾であるから，変数で始まる．導出のステップ $\gamma_i \Rightarrow \gamma_{i+1}$ は変数 A をある生成規則の本体，たとえば β に置き換える．P を構成する規則 (1) によれば，スタックの上端の A を β に置き換えることができるし，規則 (2) によればスタックの上端の終端記号と次の入力記号を照合して，一致していたら消去できる．その結果，左文形式 γ_{i+1} を表現する時点表示 $(q, y_{i+1}, \alpha_{i+1})$ に移れる.

証明を完成するには，γ_n(w に一致する) の尾は空なので，$\alpha_n = \varepsilon$ であることに注意しよう．だから $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$ であり，P は空スタックで w を受理する.

（必要性） いくらか一般化した，次のことを証明しなければならない：もし P が一連の動作によって，最終的にスタックの上端にある変数 A を，それより下は調べずにス

タックから消去した場合，文法 G は，その過程で入力から消費された記号列を，A から生成する．正確に書けば：

- もし $(q,x,A) \overset{*}{\underset{P}{\vdash}} (q,\varepsilon,\varepsilon)$ ならば，$A \overset{*}{\underset{G}{\Rightarrow}} x$.

証明は P のステップ数についての帰納法による．

基礎：1 ステップの場合．唯一の可能性は $A \to \varepsilon$ が G の生成規則で，これに対応する規則 (1) で作られた PDA P の推移規則が使われることである．この場合は $x=\varepsilon$ で，$A \Rightarrow \varepsilon$ が成り立つ．

帰納：P が 2 ステップ以上，n ステップ動いたとする．最初の動きは (1) 型で，スタックの上端の変数 A がある生成規則の本体に置き換えられる．その理由は，(2) 型の規則はスタックの上端に終端記号がある場合にしか使えないからである．使われた生成規則が $A \to Y_1Y_2\cdots Y_k$ であったとしよう．ここで各 Y_j は，終端記号か変数である．

次の $n-1$ ステップでは入力 x を消費し，Y_1Y_2 等々をスタックから一つずつ完全に消去する<**訳注**：Y_1 が別の記号列 β に書き換えられれば，見かけ上 Y_1 は消えるが，β が残っている間は「Y_1 が完全に消去された」とはいわない．β も消去されて，スタックに $Y_2\cdots Y_k$ しか残らなくなってはじめて「Y_1 が完全に消去された」という>．そこで我々は x を $x_1x_2\cdots x_k$ と分解できる：ここで x_1 は「Y_1 がスタックから消去されるまで（スタックの長さがはじめて $k-1$ なる）に消費される，入力 x の先頭の部分」を表す．また x_2 は，そのあと「Y_1 がスタックから消去されるまでに消費される，入力 x の次の部分」である（以下同様）．図 6.9 は，入力 x がどのように分解されるかと，対応するスタックの変化を示している．そこでは $Y_1Y_2\cdots Y_k$ が BaC であるとして，x を三つの部分 $x_1x_2x_3$ に分けている．一般に，もし Y_j が終端記号であれば，x_j はその終端記号 ($x_j=Y_j$) でなければならないことを注意しておこう．

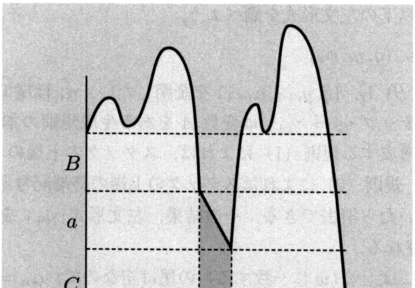


図 6.9　PDA P が x を消費し，BaC をスタックから消去するまで

以上のことは形式的に $(q, x_j x_{j+1} \cdots x_k, Y_j) \overset{*}{\vdash} (q, x_{j+1} \cdots x_k, \varepsilon)$ と書ける $(j = 1, 2, \cdots, k)$．これらのどの動作も $n-1$ ステップより短いので，帰納法の仮定から，もし Y_j が変数なら $Y_j \overset{*}{\Rightarrow} x_j$ がいえる．また Y_j が終端記号なら，一つの記号がと照合されるので，やはり $Y_j \overset{*}{\Rightarrow} x_j$ が成り立つ．この場合は導出は ε0 ステップで行われる．このように導出

$$A \Rightarrow Y_1 Y_2 \cdots Y_k \overset{*}{\Rightarrow} x_1 Y_2 \cdots Y_k \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} x_1 x_2 \cdots x_k$$

すなわち

$$A \overset{*}{\Rightarrow} x$$

が得られる．

証明を完成させるには，$A = S$，$x = w$ とおけばよい．w は $N(P)$ に属すると仮定したので，$(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$ が成り立つ．上に証明したことから $S \overset{*}{\Rightarrow} w$，すなわち w は $L(G)$ に属する． ▨

6.3.2 PDA から文法へ

今度は任意の PDA P から，それが空スタックで受理するのと同じ言語を生成する CFG G を求めてみよう．証明の背後にあるアイデアは「PDA が入力を処理する過程から，入力の一部分を消費しながら，スタックから一つの記号を完全に消去する基本過程を見定める」ことである．PDA はスタック記号を取り去るときに状態を変えるかも知れないので，一つの記号を完全に消去する間の状態の変化にも注意しなければならない．

図 6.10 は，記号列 $Y_1 Y_2 \cdots Y_k$ がスタックから消去される途中経過を示して

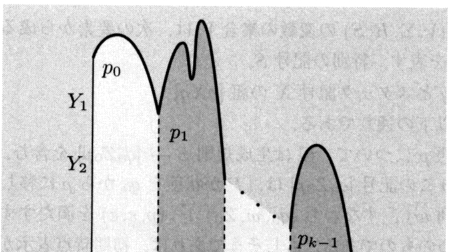


図 6.10 PDA はスタックを空にするまでに，スタック内の記号を「一つずつ，完全に」消去する．

いる．Y_1 が完全に消去される間にある入力 x_1 が消費される．「完全な消去」には何ステップもかかることがある：たとえば最初のステップで Y_1 が Z に変わり，次のステップでその Z が UV に変わり，さらにあとの過程で U が完全に消去され，それから V が完全に消去されることもある．Y_1 の完全な消去とは，ε に置き換えられることで，その間に消費されたすべての入力が x_1 を構成する．

図6.10では，状態の「基本過程」ごとの変化も示している：PDAが状態 p_0 から出発したとき，スタックの上端には Y_1 がある．この Y_1 が完全に消去されたとき，PDAは状態 p_1 にある．それからは Y_2 の消去に進むが，入力列から x_2 を読みとり（消費し），おそらく多くのステップを経た上で，スタックの上端から Y_2 を完全に消去し，状態 p_2 に移る．計算はスタックからすべての記号が除去されるまで続く．

等価な文法の構成には，次の「過程」を表現する変数を使用する．

(1)　ある記号 X がスタックから完全に消去されること，

(2)　スタックの中の X が最終的に ε に置き換えられるまでに，状態が p から q に推移すること．

そのような変数を，組み合わせ記号 $[pXq]$ で表すことにしよう．これが一つの変数を表すことに注意してほしい：文法の中で5個の記号ではなく，1個の記号として扱われる．形式的な構成法は，次の定理で示される．

定理 6.14　$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ をあるPDAとすると，$L(G) = N(P)$ を満たすある文脈自由文法 G が存在する．

証明　文法 $G = (V, \Sigma, R, S)$ の変数の集合 V は，次の要素から成る：

(1)　開始記号を表す，特別の記号 S,

(2)　状態 p，q とスタック記号 X の組 $[pXq]$

G の生成規則は以下の通りである．

a)　任意の状態 p について，G は生成規則 $S \to [q_0Z_0p]$ を含む．前に直観的に説明した通りこの記号 $[q_0Z_0p]$ は，「P が状態を q_0 から p に移し，Z_0 を消去して受理する列 w」，すなわち $(q_0, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$ を満たすすべての列 w を生成するためのものである．もしそうであれば，初期時点表示から出発した P のスタックを空にするようなすべての列 w が，出発記号 S からこれらの生成規則によって生成されることがわかる．

b)　$\delta(q, a, X)$ が対 $(r, Y_1Y_2\cdots Y_k)$ を含むとする．ここで：

　(1)　a は Σ に属する記号，あるいは $a = \varepsilon$ である．

(2)　k は 0 を含む任意の数で，$k = 0$ の場合は対は (r, ε) を表す．
すると状態の任意の列 $r_1, r_2, \cdots, r_k$ について，G は生成規則

$$[qXr_k] \ \rightarrow \ a[rY_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$$

を含む．この規則によれば，X を消去する一つの筋道は，状態 q から a(ε かも知れない）を読んでスタックの上端の X を $Y_1Y_2\cdots Y_k$ に書き換えて状態 r に移り，何かの入力を読んでスタックの Y_1 を消去して状態 r_1 に移り，それから何かの入力を読んでスタックの Y_2 を消去して状態 r_2 に移り，等々と進むことである．

次に $[qXp]$ についての上に述べた解釈が正しいことを証明しよう：

- $[qXp] \overset{*}{\Rightarrow} w$ であるための必要十分条件は，$(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$

（十分性）$(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$ と仮定する．以下 $[qXp] \overset{*}{\Rightarrow} w$ であることを，PDA のステップ数についての帰納法で証明する．

基礎：1 ステップの場合．この場合，w は一つの記号かまたは ε で，$\delta(q, w, X)$ は (p, ε) を含んでいる．G の構成法から，$[qXp] \ \rightarrow \ w$ は一つの生成規則（(b) の $k = 0$ の場合）で，$[qXp] \Rightarrow w$ が成り立つ．

帰納：$(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$ に n ステップを要し，$n > 1$ と仮定する．最初の動作は

$$(q, w, X) \ \vdash \ (r_0, x, Y_1Y_2\cdots Y_k) \ \overset{*}{\vdash} \ (p, \varepsilon, \varepsilon)$$

という形で，最初に読まれた文字を a とすると（何も読まれない場合は $a = \varepsilon$），$w = ax$ である．ここから $(r_0, Y_1Y_2\cdots Y_k)$ が $\delta(q, a, X)$ に含まれることがわかる．さらに，G の構成法から，$[qXr_k] \rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$ が生成規則であることもわかる．ここで

(1)　$r_k = p$,

(2)　$r_1, \cdots, r_{k-1}$ は P の任意の状態である．

特に，図 6.10 で示したように，記号 $Y_1, Y_2, \cdots Y_k$ は一つずつ順番に消去されるので，ちょうど Y_j が（最終的に）消去されたときの状態を r_j とすると，$(r_{k-1}, w_1, Y_j) \overset{*}{\vdash} (r_j, \varepsilon, \varepsilon)$ が成り立つ．

この推移はどれも n ステップより短いので，帰納法の仮定が使えて，$[r_{j-1}Y_jr_j] \overset{*}{\Rightarrow} w_j$ が結論できる．これらの導出を最初の生成規則につなげれば，次の結論が得られる．

$$\begin{aligned}[qXr_k] &\Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k] \\ &\overset{*}{\Rightarrow} aw_1[r_1Y_2r_2][r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \\ &\overset{*}{\Rightarrow} aw_1w_2[r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \\ &\overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} aw_1w_2\cdots w_k = w\end{aligned}$$

$r_k = p$ であるから，これで $[qXp] \overset{*}{\Rightarrow} w$ が示された．

(必要性) 証明は導出のステップ数についての帰納法で行う.

基礎: 1ステップの場合は, $[qXp] \rightarrow w$ が生成規則でなければならない. これが生成規則になるのは, P がスタックの X を消去して状態を q から p に変える推移がある場合, すなわち $w = a$ で $\delta(q, a, X)$ が (p, ε) を含む場合に限る. そしてそれなら $(q, w, X) \vdash (p, \varepsilon, \varepsilon)$ が成り立つ.

帰納: $[qXp] \stackrel{*}{\Rightarrow} w$ が2ステップ以上, n ステップで成り立つと仮定する. その最初のステップは次のような形をしているはずである:

$$[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k] \stackrel{*}{\Rightarrow} w$$

ここで $r_k = p$ である. これが可能であるためには, $\delta(q, a, X)$ が $(r_0, Y_1Y_2\cdots Y_k)$ を含んでいなければならない.

さて w は, $[r_{j-1}Y_jr_j] \stackrel{*}{\Rightarrow} w_j$ を満たす w_j ($j = 1, 2, \cdots, k$) について, $w = aw_1w_2\cdots w_k$ のように分解できる. 帰納法の仮定から, すべての j について

$$(r_{j-1}, w_j, Y_j) \stackrel{*}{\vdash} (r_j, \varepsilon, \varepsilon)$$

が成り立つ. ここで定理6.5を使えば, w_j より先の正しい入力, Y_j より下の正しいスタック記号をつけ加えて, 次の導出を導くことができる.

$$\begin{aligned} &(r_{j-1}, w_jw_{j+1}\cdots w_k, Y_jY_{j+1}\cdots Y_k) \\ \stackrel{*}{\vdash}\ &(r_j, w_{j+1}\cdots w_j, Y_{j+1}\cdots Y_k) \end{aligned}$$

これらの導出を全部つなげば, 次のようになる.

$$\begin{aligned} &(q, aw_1w_2\cdots w_k, X) \stackrel{*}{\vdash} (r_0, w_1w_2\cdots w_k, Y_1Y_2\cdots Y_k) \\ \stackrel{*}{\vdash}\ &(r_1, w_2w_3\cdots w_k, Y_2Y_3\cdots Y_k) \\ \stackrel{*}{\vdash}\ &(r_2, w_3\cdots w_j, Y_3\cdots Y_k) \\ \stackrel{*}{\vdash}\ &\cdots \stackrel{*}{\vdash} (r_k, \varepsilon, \varepsilon) \end{aligned}$$

$r_k = p$ であるから, これで $(q, w, X) \stackrel{*}{\vdash} (r_k, \varepsilon, \varepsilon)$ が証明できた.

証明は次のようにして完成される. $S \stackrel{*}{\Rightarrow} w$ となるのは, 開始記号 S にかかわる生成規則の作り方から, ある状態 p について $[q_0Z_0p] \stackrel{*}{\Rightarrow} w$ となる場合である. さきほど証明したことによれば, $[q_0Z_0p] \stackrel{*}{\Rightarrow} w$ となる必要十分条件は $(q_0, w, X) \stackrel{*}{\vdash} (p, \varepsilon, \varepsilon)$ が成り立つ, すなわち P が空スタックで w を受理することである. したがって $L(G) = N(P)$ ▨

例 6.15 例6.10の文法から構成したPDA $P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$ を考えてみよう. P_N は,「どの e (else) も, それより前の i (if) に対応している」という規則を(その右端で, はじめて)破るような列全体の集合である. P_N の状態は一つの状態と一つのスタック記号しか持たないので, 対応する文

法の構成はきわめて簡単である．文法 G の変数は，次の二つしかない：

a) 開始記号 S——これは定理 6.14 の方法で構成されるどんな文法にも含まれる．

b) $[qZq]$——P_N の状態とスタック記号から作れる，この場合ただ一つの組み合わせ記号である．

文法 G の生成規則は以下の通りである．

(1) S に対する唯一の生成規則は $S \to [qZq]$．もし PDA が n 個の状態を持つなら，最後は n 個の状態のどれでもよいので，この型の生成規則は n 個になる．この例のように，最初の状態は開始状態，スタック記号は開始記号でなければならない．

(2) $\delta_N(q,i,Z)$ は (q,ZZ) を含むから，生成規則 $[qZq] \to i[qZq][qZq]$ が得られる．今の場合，この型の生成規則も一つしかないが，状態が n 個ある場合には，本体は任意の中間状態を持てるので，規則は n 個に増える．すなわち PDA の状態 r に対して，$[qZq] \to i[qZr][rZq]$ が生成規則になる．

(3) $\delta_N(q,e,Z)$ は (q,ε) を含むので，生成規則

$$[qZq] \to e$$

が得られる．この場合，Z は空列に置き換えられるので，本体の唯一の記号はこの動作を引き起こす入力記号である．

簡単のために，組み合わせ記号 $[qZq]$ をより簡単な記号，たとえば A で表すことにしよう．するとこの文法は，次の生成規則だけから成る：

$$S \to A$$

$$S \to iAA \mid e$$

実は，A も S もまったく同じ列を生成するので，同一視してよい．だから文法は，次の 1 行で完全に記述できる：

$$G = (\{S\}, \{i,e\}, \{S \to iSS \mid e\}, S)$$ ▨

6.3.3 6.3 節の演習問題

♣ **問 6.3.1** 次の文法を，同じ言語を空スタックで受理する PDA に変換せよ．

$$\begin{aligned} S &\to 0S1 \mid A \\ A &\to 1A0 \mid S \mid \varepsilon \end{aligned}$$

♣ **問 6.3.2**　次の文法を，同じ言語を空スタックで受理する PDA に変換せよ．

$$S \rightarrow aAA$$
$$A \rightarrow aS \mid bS \mid a$$

♣ * **問 6.3.3**　PDA $P = (\{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$ を CFG に変換せよ．ただし δ は次のように定義されるとする．

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$.
2. $\delta(q, 1, X) = \{(q, XX)\}$.
3. $\delta(q, 0, X) = \{(p, X)\}$.
4. $\delta(q, \varepsilon, X) = \{(q, \varepsilon)\}$.
5. $\delta(p, 1, X) = \{(p, \varepsilon)\}$.
6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$.

♣ **問 6.3.4**　問 6.1.1 の PDA を文脈自由文法に変換せよ．

♣ **問 6.3.5**　以下の文脈自由言語のそれぞれに，同じ言語を空スタックで受理する PDA を作れ．まず文法を構成し，それを PDA に変換してもよい．

a)　$\{a^m b^n c^{2(m+n)} \mid n \geq 0, m \geq 0\}$.

b)　$\{a^i b^j c^k \mid i = 2j$ または $j = 2k\}$.

!c)　$\{0^n 1^m \mid n \leq m \leq 2n\}$.

♣ **問 6.3.6**　もし P が PDA ならば，$N(P_1) = N(P)$ であるような 1 状態の PDA P_1 が存在することを示せ．

♣ ! **問 6.3.7**　s 状態で t 個のスタック記号を持つ PDA が，スタックの記号を書き換えるどの規則も u より長い列は使わない，とする．この PDA を 6.3.2 項の方法で CFG に変換したとき，現れる変数の個数の正確な上限を示せ．

6.4　決定性プッシュダウン・オートマトン

PDA の定義では，非決定性の（非決定論的な）動作が許されているが，決定性の場合はとても重要である．特にパーサーはふつう，決定性の PDA のように振舞う．だからそのようなオートマトンで受理できる言語のクラスは，プログラミング言語への応用にどんな構造が適しているのかについて見通しを与えてくれる点で，興味深い．この節では，決定性 PDA を定義し，それに何ができ何ができないかを，いくつかの例について調べる．

6.4.1 決定性 PDA の定義

直観的にいって，ある PDA が決定性であるとは，どんな状況でも動作の選択肢がない（動作が一通りにきまる）ことである．選択肢には，二つの種類がある．もし $\delta(q,a,X)$ が二つ以上の対を含むなら，PDA は次の動作としてそれらの対のどちらを選んでもよいので，もちろん非決定性である．しかし，$\delta(q,a,X)$ がいつも一つの要素しか含まない（か空集合である）としても，現実の入力記号を消費するか ε-動作を行うかの選択があり得る．そこで PDA $P=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ が**決定性**である（決定性 PDA，あるいは DPDA である）ことを，次の二つの条件によって定義する：

(1) Q の任意の要素 q，Σ の任意の要素 a あるいは $a=\varepsilon$，および Γ の任意の要素 X に対して，$\delta(q,a,X)$ は高々一つの要素を含む．

(2) もし Σ の中のある a に対して，$\delta(q,a,X)$ が空でなければ，$\delta(q,\varepsilon,X)$ は空である．

例 6.16 例 6.2 の言語 L_{wwr} は，CFL であるが DPDA では受理できないことが知られている．しかし「中心の目印」c を真中におくと，DPDA で認識できる言語になる．すなわち，言語 $L_{wcwr}=\{wcw^R \mid w \text{ は } (\mathbf{0}+\mathbf{1})^* \text{に属する}\}$ は決定性 PDA で認識できる．

その DPDA の作戦は，中央の目印 c に到達するまで，0 と 1 をスタックに記録し，それから別の状態に移って，入力記号とスタックの上端とを照合する．そしてもし一致したらスタックの上端の記号を消去し，一致しなかったらそこで終わる：あとの場合，入力列は wcw^R の形ではあり得ない．もしスタックの底にある開始記号が露わになるまで，他のスタック記号の消去に成功したら，入力列を受理する＜**訳注**：入力列がそこで終わらなかった場合には，余分な入力記号をスタックの開始記号と照合することになり，受理されない＞．

このアイデアは，図 6.2 に示した PDA とよく似ている．しかしあの PDA は，状態 q_0 で入力記号をスタックに押し込んでもよいし，そこが列の中央かと推測して，ε-動作で状態 q_1 に移ってもよいので，非決定性である．L_{wcwr} のための DPDA の推移図を図 6.11 に示した．

この PDA は，明らかに決定性である．同じ状態・同じ入力記号・同じスタック記号に対する動作の選択肢はない．実際の入力記号を使うか ε を使うかの選

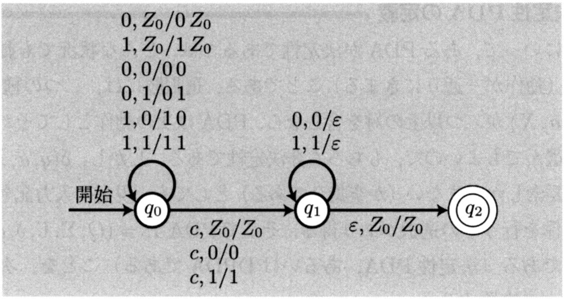


図 6.11 L_{wcwr} を受理する決定性 PDA

択についても，ε 動作はスタックの上端が Z_0 で状態 q_1 から q_2 に移るときしかないが，状態 q_1 でスタックの上端に Z_0 がある場合には，他の動作はできない．▨

6.4.2 正則言語と決定性 PDA

DPDA が受理する言語のクラスは，正則言語と CFL の中間に位置する．まず DPDA 言語がすべての正則言語を含むことを証明しよう．

定理 6.17 もし L が正則言語ならば，ある DPDA P について $L = L(P)$ が成り立つ．

証明 本質的に，DPDA は決定性有限オートマトンのシミュレーションができる．PDA はスタックを持っているので，何かスタック記号 Z_0 をスタックに置くが，それを無視して状態だけで動作を決めればよい．正確にいうために，$A = (Q, \Sigma, \delta_A, q_0, F)$ を DFA として，DPDA を次のように構成する：

$$P = (Q, \Sigma, \{Z_0\}, \delta_F, q_0, Z_0, F)$$

ここで Q のすべての状態 p，q に対して，もし $\delta_A(q, a) = p$ であれば $\delta_F(q, a, Z_0) = \{(p, Z_0)\}$ と定義する．

証明したいことは，「$(q_0, w, Z_0) \overset{*}{\vdash} P(p, \varepsilon, Z_0)$ となる必要十分条件が $\hat{\delta}_A(q_0, w) = p$ となること」である．すなわち，P は状態によって A のシミュレーションを行う．証明は両方向とも $|w|$ についての帰納法で容易にできるから，読者にお任せする．A も P も，F の状態のどれかに達したときに受理するので，両者の言語は一致する．▨

空スタックで受理するDPDAは，言語を認識する能力がかなり制限される．**プレフィックス性**を持つ言語L，すなわちLのどの列x, y ($x \neq y$) も「xがyのプレフィックス**ではない**」という言語を考えてみよう．

例 6.18 例6.16の言語L_{wcwr}はプレフィックス性を持っている．すなわち，二つの列wcw^Rとxcx^Rは，一致する場合を除いて，一方が他方のプレフィックスになることはあり得ない．実際，もし列wcw^Rがxcx^Rのプレフィックスでしかも$w \neq x$であったと仮定すると，wはxより短い．したがってwcw^Rの中のcは，xの中の0または1にあたることになる．cは0でも1でもないのでそれは不可能で，最初の仮定は誤りである．

一方，プレフィックス性を持たないごく簡単な言語がある．たとえば0の列全体の集合$\{0\}^*$は，その中のどの二つも一方が他方のプレフィックスになっている．だからこの言語はプレフィックス性を持っていない．実際，「**任意**の二つの列の一方が他方のプレフィックスになっている」ことは，プレフィックス性が成り立たないことを示す必要以上に強い条件である．

$\{0\}^*$は正則言語である．そこから「すべての正則言語があるDPDAによって空スタックで受理できる」ことも**正しくない**ことがわかる．その関係を示す次の定理は，演習として残しておこう：

定理 6.19 言語LがあるDPDAによって空スタックで受理される必要十分条件は，その言語がプレフィックス性を持ち，しかもあるDPDAによって最終状態で受理されることである． ▨

6.4.3 DPDAと文脈自由言語

DPDAがL_{wcwr}のような言語を受理できることはすでに述べた．L_{wcwr}は正則ではないが，そのことをたしかめるには，正則であると仮定して，反復補題を使うとよい．nを反復補題の定数として，L_{wcwr}に属する列$w = 0^n c 0^n$を考える．この列に「反復」を適用すると，前半の0の数だけが増えてしまい，中央の目印cが中央でなくなる．それではL_{wcwr}に属さないので，矛盾であり，L_{wcwr}は正則言語ではないと結論できる．

一方，L_{wwr}のように，CFLではあるが「どんなDPDA Pによっても，最終状態でも受理できない」言語が存在する．厳密な証明は煩雑であるが，直観

的には明らかである．もし DPDA P が L_{wwr} を受理したとすれば，与えられた 0 の列をスタックに記録するか，あるいは「いくらでも大きな数を数える」何かの手段を講じなければならない．たとえば 0 を二つ読むごとに X を一つスタックに記録し，奇数か偶数かは状態で区別してもよい．

P が n 個の 0 を読み，そのあと 110^n を読んだとする．11 のあとに n 個の 0 があることを確かめるには，スタックを空にしなければならない[5)]．さて，P は 0^n110^n を読んだ．そのあと続けて同じ列が来ると，全体として ww^R の形になるから，P はそれを受理しなければならない．しかしスタックが空なので，任意の数 n が何であったのかは記録されていない．だから L_{wwr} を正しく受理することはできない．我々の結論は：

- DPDA によって最終状態で受理できる言語のクラスは，正則言語のクラスを真に含んでいるが，CFL のクラスに真に含まれている．

6.4.4 DPDA とあいまいな文法

DPDA の能力は，あいまいでない文法と比較するともっとよくわかる．残念ながら，DPDA は「本質的にあいまいでない文脈自由言語」の部分クラスである．たとえば言語 L_{wwr} は，あいまいでない文法

$$S \to 0S0 \mid 1S1 \mid \varepsilon$$

によって記述できるが，DPDA では受理できない．次の二つの定理は，上に黒丸つきで述べたことをさらに明確にしてくれる．

定理 6.20 ある DPDA P について $L = N(P)$ となるならば，L はあるあいまいでない文脈自由文法で記述できる．

証明 P が決定性ならば，定理 6.14 で示した構成法によって得られる CFG G があいまいでないことを証明すればよい．定理 5.29 から，その文法の最左導出が一意的であることを示せばよい．

P が列 w を空スタックで受理すると仮定する．そのときの P の動作は，決定性なのだから，一意的であって，スタックが空になればもう動けない．その動作から，G が w を生成する最左導出が求まる．P の規則から，使われる生成規則の選択肢が生まれることはけっしてない．しかし P の規則，たとえば $\delta(q, a, X) = \{(r, Y_1Y_2\cdots Y_k)\}$

---

5) ここが厳密に証明しようとすると，非常にむずかしいところである．P は他の方法で，0 の個数を比較できないだろうか？

は，P が $Y_1, Y_2, \cdots, Y_{k-1}$ のそれぞれをスタックから消去したあとに到達する状態によって，G のいろいろな生成規則に対応し得る．しかし P は決定性であるから，そのうちの一つだけが P の実際の動作と適合するので，それらの生成規則の一つだけが w の生成に実際に成功する． ▨

実はもっと強く,「DPDA が最終状態で受理する言語も，あいまいでない文法で記述できる」ことも証明できる．我々が知っている「PDA から文法へ」の構成法は空スタックで受理する場合であるから，まず言語をプレフィックス性を持つように修正し，それを受理する DPDA からあいまいでない文法を構成し，それからその文法をもとの言語を生成するように修正すればよい.「プレフィックス性を持つ」ようにするには，終端記号をつけ加えればよい．

定理 6.21 ある DPDA P について $L = L(P)$ となるならば，L はあるあいまいでない文脈自由文法で記述できる．

証明 \$ を，L では使われていない記号とし，これを「末尾記号」として使用する．つまり L の各列の末尾に \$ をつけ加えたものの全体を L' とおく：$L' = L\$$. すると L' はたしかにプレフィックス性を持つので，定理 6.19 によれば $L' = N(P')$ を満たす DPDA P' が存在する[6]．そこで定理 6.20 を使うと，$L' = N(P')$ を生成するあいまいでない文法 G' がある．

次に，G' から $L(G) = L$ を満たす文法 G を構成しよう．それにはただ，各列から末尾記号 \$ を除去すればよい．だから \$ を G の変数と見て，生成規則 $\$ \to \varepsilon$ を導入すればよい．それ以外は，G' の生成規則をそのまま G でも採用する．$L(G') = L'$ であるから，もちろん $L(G) = L$ である．

この G はあいまいでない．それを示すには，G の最左導出と G' の最左導出が，最後を除いて一致することに注意するとよい：G の最左導出は，最後に \$ を ε に置き換えるところだけが違っている．だからもし w を生成する G の二つの最左導出があれば，$w\$$ を生成する G' の二つの最左導出がある．しかし G' はあいまいでないので，G もあいまいでない． ▨

---

[6] 定理 6.19 の証明は問 6.4.3 にあるが，P から P' を構成するのは容易である．新しい状態 q をつけ加えて，P が受理状態に入ったときに P' は q に入り，次の入力が \$ であれば受理する．状態 q では，P' はスタックのすべての記号を消去する．また P' のスタックの底には独自の開始記号をおいて，P を模倣する途中で偶然にスタックが空になるのを防ぐようにする．

6.4.5 6.4 節の演習問題

♣ **問 6.4.1** 次の PDA は決定性か．DPDA の定義を満たすことを示すか，それを破る規則を指摘せよ．

a) 例 6.2 の PDA

*b) 問 6.1.1 の PDA

c) 問 6.3.3 の PDA

♣ **問 6.4.2** 次の言語を受理する決定性プッシュダウン・オートマトンを作れ．

a) $\{0^n 1^m \mid n \leq m\}$

b) $\{0^n 1^m \mid n \geq m\}$

c) $\{0^n 1^m 0^n \mid n$ と m は任意 $\}$

♣ **問 6.4.3** 定理 6.19 は，次の 3 段階に分けて証明できる：

*a) もしある DPDA P について $L = N(P)$ となるならば，L はプレフィックス性を持つ．

!b) もしある DPDA P について $L = N(P)$ となるならば，ある DPDA P' があって $L = L(P')$ を満たす．

*!c) もし L がプレフィックス性を持ち，しかもある DPDA P' について $L(P')$ となるならば，ある DPDA P があって $L = N(P)$ を満たす．

♣ **問 6.4.4** 言語

$$L = \{0^n 1^n \mid n \geq 1\} \cup \{0^n 1^{2n} \mid n \geq 1\}$$

が文脈自由言語であること，またどんな DPDA でも受理できないことを証明せよ．**ヒント**：0^n という形の列で，n の二つの異なる値（たとえば n_1 と n_2）に対して，L を受理すると仮定された DPDA が，それらを読み込んだあとで同じ時点表示になることを示す．直観的にはその DPDA は，0 を読んだときにスタックに記録したものを，1 を読むときに（同じ数かどうかを確かめるために）あらかた消去してしまう．だからその DPDA は，そのあとで n_1 個の 1 を読んだときに受理すべきか n_2 個の 1 を読んだときに受理すべきかが，わからなくなってしまう．

6.5 第 6 章の要約

◆ **プッシュダウン・オートマトン**：PDA とは，任意の長さの列を記録できる「スタック」を装備した，非決定性の有限オートマトンのことである．スタックの読み・書きは，その上端でしかできない．

◆ **プッシュダウン・オートマトンの動作**：PDA は次の動作を，① 現在の状態，② 次の入力記号，③ スタックの上端にある記号の組み合わせに基づいて選択する．また入力記号を読まず（消費せず）に，それと関係なくある動作を行うこともある．非決定性であるから，PDA はいくつか有限個の動作から一つを選ぶことができる．ど

の動作でも，新しい状態に進みスタックの上端をある記号列に置き換える．

◆**プッシュダウン・オートマトンによる受理**：PDA が受理を合図する二つの方式がある．一つは受理状態に入ることで，もう一つはスタックを空にすることである．これらの方式は，一方の方式で受理される言語はもう一つの方式でも（ある別の PDA によって）受理されるという意味で，等価である．

◆**時点表示**：ある PDA の「現在の状況」を表現するために，状態，入力の残り，およびスタックの内容から成る時点表示が使われる．時点表示の間の推移 $\vdash$ は，PDA の 1 ステップの動作を表す．

◆**プッシュダウン・オートマトンと文法**：PDA が最終状態あるいは空スタックによって受理する言語のクラスは，文脈自由言語のクラスにぴったり一致する．

◆**決定性プッシュダウン・オートマトン**：PDA は，どんな状態，(ε を含む）入力，スタック記号に対しても，次の動作が一意的に決まる場合，決定性といわれる．入力記号をほんとうに読むか ε 入力で動くかも，一意的でなければならない．

◆**決定性プッシュダウン・オートマトンの受理**：最終状態による受理と空スタックによる受理は DPDA の場合，一般の PDA とは違って，等価にはならない．空スタックで受理される言語は，最終状態で受理されるもののうち，プレフィックス性を持つ（その言語のどの列も，同じ言語内の他の列のプレフィックスでない）ものに限る．

◆**DPDA が受理する言語**：すべての正則言語は，ある DPDA によって最終状態で受理される．また DPDA は，正則でない言語を受理できる．DPDA 言語は文脈自由言語であるが，実はあいまいでない CFG で記述できるものに限る．このように DPDA の言語は，正則言語と文脈自由言語の中間に位置している．

6.6　第 6 章の参考文献

プッシュダウン・オートマトンのアイデアは，A. G. Oettinger [4] と M. P. Schutzenberger [5] に帰せられている．プッシュダウン・オートマトンと文脈自由言語との等価性は，やはり独立に発見された：N. Chomsky の 1961 年の MIT 技術報告に含まれているが，公刊は J. Evey [1] が最初である．

決定性 PDA は，P. C. Fischer [2] と M. P. Schutzenberger [5] によってはじめて導入された．これはのちにパーサーのモデルとして，役立てられた．特に，D. E. Knuth [3] の「LR(k) 文法」は，CFG の部分クラスで，ちょうど DPDA 言語を生成する．この LR(k) 文法はまた，5.3.2 項で紹介したパーサーを生成するツール，YACC の基礎を与える．

[1] J. Evey, “Application of pushdown store machines,” *Proc. Fall Joint Computer Conference* (1963), AFIPS Press, Montvale, NJ, pp.215–227.

[2] P. C. Fischer, “On computability by certain classes of restricted Turing

machines," *Proc. Fourth Annl. Symposium on Switching Circuit Theory and Logical Design* (1963), pp.23–32.

[3] D. E. Knuth, "On the translation of languages from left to right," *Information and Control* **8**:6 (1965), pp.607–639.

[4] A. G. Oettinger, "Automatic syntactic analysis and the pushdown store," *Proc. Symposia on Applied Math.* **12** (1961), American Mathematical Society, Providence, RI.

[5] M. P. Schutzenberger, "On context-free languages and pushdown automata," *Information and Control* **6**:3 (1963), pp.246–264.

7 文脈自由言語の性質

文脈自由言語の学習を，その性質をいくつか調べることによって完成させよう．最初の課題は文脈自由文法の単純化である．ある言語が CFL なら，それはある単純な形の文法で生成されることがいえ，単純化は CFL に関する命題の証明をやさしくする．

次に CFL の反復補題を証明する．この定理は，正則言語に対する定理 4.1 と同じ意味を持ち，ある言語が文脈自由でないことの証明に使える．さらに，4 章で正則言語について調べたのと同様の閉包性や決定問題を調べる．正則言語で成り立つ閉包性のいくつか (全部ではない) は文脈自由言語でも成り立つ．同様に，CFL に対する決定問題のいくつかは，正則言語に対して開発された決定法を拡張することによって，決定できる．しかし，答えることが不可能な CFL の決定問題もある．

7.1 文脈自由言語の標準形

この節の目標は，すべての (ε を含まない) CFL は，生成規則がすべて，$A \to BC$ または $A \to a$ (A, B, C は変数，a は終端記号) の形をした CFG で生成されることを示すことである．この形は**チョムスキー** (Chomsky) **標準形**と呼ばれる．チョムスキー標準形に到達するために，多くの準備的な単純化を必要とするが，それらは，それ自身いろいろなところで役立つ．

(1) **無用な記号** (useless symbol) の除去：無用な記号とは，開始記号から終端記号列を導出する際に決して現れない変数や終端記号である．

(2) **ε-規則** (ε-production) の除去：$A \to \varepsilon$ (A は変数) の形の規則を除去する．

(3) **単位規則** (unit production) の除去：単位規則は $A \to B$ (A, B は変数) の形の規則である．

7.1.1 無用な記号の除去

記号 X が文法 $G = (V, T, P, S)$ で**有用** (useful) であるというのは，導出 $S \Rightarrow \alpha X \beta \overset{*}{\Rightarrow} w$ (w は T^* の列) が存在する場合をいう．ここで X は V の元でも T の元でもよい．また文形式 $\alpha X \beta$ は導出の最初の S でも，最後の w でもよい．X が有用でないとき，**無用** (useless) であるという．無用な記号を省いても生成される言語は変わらないので，無用な記号をすべて検出し除去した方がよい．

無用な記号の除去への入り口は，記号が無用になる二つの状況を区別することである．

(1) X が**生成的** (generating) であるというのは，$X \overset{*}{\Rightarrow} w$ (w は終端記号列) となるときである．すべての終端記号は生成的であることに注意せよ．それらは 0 ステップで自分自身を導出する．

(2) X が**到達可能** (reachable) であるというのは，$S \overset{*}{\Rightarrow} \alpha X \beta$ となるときである．

有用な記号は明らかに生成的かつ到達可能である．最初に生成的でない記号を除去し，得られた文法からさらに到達可能でない記号を除去すれば，以下で証明されるように，有用な記号だけが残る．

例 7.1　次の文法を考えよう．

$$S \to AB \mid a$$
$$A \to b$$

B 以外の記号は生成的である．a と b は自分自身を生成し，S は a を A は b を生成する．B を除去するときは，同時に規則 $S \to AB$ も除去しなければならず，次の規則が残る．

$$S \to a$$
$$A \to b$$

ここで S と a だけが S から到達可能だとわかる．A と b を除去し，規則 $S \to a$ だけが残る．この規則はそれ自身，もとの文法で生成される言語 $\{a\}$ を生成す

る文法である．

到達可能性の検査から先に始める場合，文法

$$S \to AB \mid a$$
$$A \to b$$

中の記号がすべて到達可能であることに注意しよう．次に生成的でない記号 B を除去してもまだ無用な記号 A と b が残ってしまう．▨

定理 7.2　$G = (V, T, P, S)$ を文脈自由文法とし，$L(G) \neq \varnothing$，すなわち，少なくとも一つは列を生成するものとする．$G_1 = (V_1, T_1, P_1, S)$ を次の手順で得られる文法とする．

(1)　生成的でない記号とそれらを一つでも含む規則を除去する．$G_2 = (V_2, T_2, P_2, S)$ をこうして得られた新しい文法とする．$L(G)$ は空でないと仮定したので，S は生成的であり，除去されない．

(2)　G_2 で到達可能でない記号を除去する．

G_1 は無用な記号を持たず，$L(G_1) = L(G)$ である．

証明　記号 X が残った，すなわち $X \in V_1 \cup T_1$ とする．X は第 1 ステップで除去されなかったから G で生成的であり，ある $w \in T^*$ に対して $X \overset{*}{\underset{G}{\Rightarrow}} w$ が成り立つ．この導出は，現れる記号がすべて生成的であるから，G_2 の導出でもあり，$X \overset{*}{\underset{G_2}{\Rightarrow}} w$ である．

さらに X は第 2 ステップでも除去されなかったから，G_2 で到達可能であり，$S \overset{*}{\underset{G_2}{\Rightarrow}} \alpha X \beta$ となる α, β が存在する．この導出中に現れる記号はすべて G_2 で到達可能であるから，$S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta$ である．

したがって，$\alpha X \beta$ に現れる記号はすべて G_1 で到達可能であり，しかも $V_2 \cup T_2$ に属するので G_2 で生成的であり，ある終端記号列を導出する．さらに，G_2 における $\alpha X \beta$ からの終端記号列の導出 $\alpha X \beta \overset{*}{\underset{G_2}{\Rightarrow}} xwy$ に含まれる記号はすべて S から到達可能である．したがって，この導出はまた G_1 の導出でもある．すなわち，

$$S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta \overset{*}{\underset{G_1}{\Rightarrow}} xwy$$

結局，X は G_1 でも有用である．X は G_1 の任意の記号だったので，G_1 は無用な記号を持たない．

最後に $L(G_1) = L(G)$ を示す．常道にしたがい，二つの集合が等しいことを示すために，各々が他方に含まれることを示す．

$L(G_1) \subseteq L(G)$：G から G_1 を構成する際には，記号や規則を除去するだけなので，$L(G_1) \subseteq L(G)$ である．

$L(G) \subseteq L(G_1)$：任意の列 w は，$L(G)$ の元ならば $L(G_1)$ の元でもあることを示す．w が $L(G)$ の元なら $S \overset{*}{\underset{G}{\Rightarrow}} w$ である．この導出に現れる記号は明らかにすべて生成的かつ到達可能であるから，それは G_1 の導出でもある．すなわち，$S \overset{*}{\underset{G_1}{\Rightarrow}} w$ であり，w は $L(G_1)$ に属する． ▨

7.1.2 生成的記号と到達可能記号の計算

議論し残した点が二つある．生成的記号の集合と到達可能記号の集合をそれぞれどのように計算するかという点である．二つの問題に対して，使われるアルゴリズムはこれらの記号を見つけるために最善を尽くしていることを示そう．すなわち，これらの集合の正しい帰納的構成法になっていて，ある記号が生成的 (または到達可能) であると決定できなかったときには，実際にそうでないことを証明する．

$G = (V, T, P, S)$ を文法とする．G の生成的記号を計算するには，次の帰納的構成法を実行する．

基礎：T の元はすべて自分自身を生成するので，生成的記号の集合に加える．

帰納：規則が $A \to \alpha$ でかつ α 中に現れる記号がすべて生成的記号の集合に属せば，A も生成的記号の集合に加える．このステップは $\alpha = \varepsilon$ の場合も含むので，右辺が ε である規則を持つ変数はすべて生成的記号の集合に加えられる．

例 7.3 例 7.1 の文法を考えよう．基底ステップにより，a と b は生成的である．帰納ステップにおいて，規則 $A \to b$ を使えば A が生成的記号の集合に加わり，規則 $S \to a$ を使えば S が加わる．ここで帰納ステップは終わる．B が生成的記号の集合に入っていないので，規則 $S \to AB$ は使えない．したがって生成的記号の集合は $\{a, b, A, S\}$ である． ▨

定理 7.4 上記のアルゴリズムは G の生成的記号の集合を正しく計算する．

証明 アルゴリズムで決定された記号が実際に生成的であることの証明は，生成的記号の集合に加わる順番に関する帰納法で容易に証明できる．この証明は，読者への演習問題として残しておこう．もう一方向の証明のために，X を生成的記号とし，$X \overset{*}{\underset{G}{\Rightarrow}} w$ とする．この導出のステップ数に関する数学的帰納法で，X が (上記のアルゴリズムで) 生成的記号の集合に加わることを証明する．

基礎：導出ステップ数が 0 の場合，X は終端記号で，アルゴリズムの基底ステップで生成的記号の集合に含まれる．

帰納：導出に n ステップ $(n > 0)$ かかるとき，X は変数である．導出の最初に使われる規則が $X \to \alpha$ で，$X \Rightarrow \alpha \overset{*}{\Rightarrow} w$ であるとする．α 中に現れる記号は，すべて w の一部である記号列（空列かもしれない）を生成し，その導出のステップ数はすべて n より小さい．帰納法の仮定から，α に現れる記号はすべて生成的記号の集合に属すので，アルゴリズムの帰納ステップは，規則 $X \to \alpha$ を使って X を生成的記号の集合に加える．▨

さて，文法 $G = (V, T, P, S)$ の到達可能記号の集合を見つける帰納的アルゴリズムを考えよう．再び，アルゴリズムは到達可能記号を発見することに最善を尽くしていることを示すことができる．

基礎：S を到達可能記号の集合に加える．

帰納：A が到達可能記号の集合に属していれば，A を頭部に持つ規則の本体に現れる記号をすべて到達可能記号の集合に加える．

例 7.5　再び，例 7.1 の文法を扱う．基底ステップにより，S を到達可能記号の集合に加える．S の規則は右辺に AB と a を持つので，A, B, a も到達可能記号の集合に加える．B は規則を持たず，A の規則は $A \to b$ のみであるので，b が加わる．もうこれ以上加わる記号はなく，到達可能集合は $\{S, A, B, a, b\}$ になる．

定理 7.6　上記のアルゴリズムは G の到達可能記号の集合を正しく計算する．

証明　証明は定理 7.4 と同様の数学的帰納法による．読者への演習問題として残しておこう．▨

7.1.3 ε-規則の除去

ε-規則は，文法の設計上は非常に便利であるが，本質的でないことを示そう．もちろん，本体が ε の規則を持たない文法では ε を導出することは不可能である．したがって，実際に証明するのは，「言語 L が CFG ならば，$L - \{\varepsilon\}$ を生成する ε-規則を持たない CFG が存在する」ということである．ε が L に属さなければ，$L = L - \{\varepsilon\}$ なので，L は ε-規則を持たない文法を持つ．

方針は，「消去可能」な変数を発見することである．変数は $A \overset{*}{\Rightarrow} \varepsilon$ のとき，**消去可能** (nullable) であるという．A が消去可能であるとする．A が規則の本体に，たとえば $B \to CAD$ のように現れたとき，A は ε を導出する可能性があ

る (しない可能性もある)．そこでこの規則から二つの規則を作る．一つは，右辺から A を除いた規則 $(B \to CD)$ であり，A から ε を導出した場合に対応する．もう一つは，A が残っている規則 $(B \to CAD)$ である．しかし，A が残っている規則を使ったときには，A から ε が導出できないようにしなければならない．それには，単に ε を本体に持つ規則をすべて除いて，どの変数からも ε が導出されないようにする．

$G = (V, T, P, S)$ を文脈自由文法とする．G のすべての消去可能変数を次の帰納的アルゴリズムで決定することができる．さらに，アルゴリズムが決定する以外には消去可能変数がないことも示そう．

基礎：$A \to \varepsilon$ が G の規則ならば，A は消去可能である．

帰納：$B \to C_1C_2 \cdots C_k$ が G の規則で，各 C_i がすべて消去可能ならば，B は消去可能である．各 C_i は消去可能な変数でなければならないので，本体がすべて変数から成る規則だけ検査すればよいことに注意せよ．

定理 7.7　任意の文法 G において，G の消去可能変数は，上記のアルゴリズムが決定するものと一致する．

証明　定理の主張は「A が消去可能である必要十分条件は，アルゴリズムが A を消去可能と決定したときである」と表せる．十分条件であることは，アルゴリズムで消去可能な変数を見つける順番に関する帰納法で，それらが本当に空列を生成することを示せばよい．必要条件であることは最短導出 $A \overset{*}{\Rightarrow} \varepsilon$ の長さに関する帰納法で以下のように証明する．

基礎：導出ステップ数が 1 のとき，$A \to \varepsilon$ は G の規則でなければならず，A はアルゴリズムの基底ステップで，消去可能であると決定される．

帰納：導出 $A \overset{*}{\Rightarrow} \varepsilon$ の長さが n $(n > 1)$ であるとする．導出は，$A \Rightarrow C_1C_2 \cdots C_k \overset{*}{\Rightarrow} \varepsilon$ の形をしているはずである．ここで，各 C_i は長さ n 未満の導出で ε を導出する．帰納法の仮定により，アルゴリズムは各 C_i を消去可能であると決定する．よって，アルゴリズムの帰納ステップにより，規則 $A \to C_1C_2 \cdots C_k$ を適用して，A は消去可能と決定される．　▨

ε-規則なしの文法を構成しよう．$G = (V, T, P, S)$ を CFG とする．G の消去可能変数を決定し，P_1 が次で与えられる文法 $G_1 = (V, T, P_1, S)$ を構成する．

$A \to X_1X_2 \cdots X_k$ $(k \geq 1)$ を P に属する規則とし，k 個の X_i のうちの m 個が消去可能だとする．新しい文法 G_1 は消去可能な変数それぞれのあるなしに対

応したすべての組み合わせの 2^m 個の規則を持つ．例外が一つある．$m = k$，つまりすべての X_i が消去可能なとき，すべてを空列に置き換えて得られる $A \to \varepsilon$ は除く．さらに，P にもともと属していた $A \to \varepsilon$ も P_1 には加えない．

例 7.8　次の文法を考える．

$$S \to AB$$

$$A \to aAA \mid \varepsilon$$

$$B \to bBB \mid \varepsilon$$

最初に消去可能変数を求める．A と B は，ε-規則を持つので直接消去可能である．次に，$S \to AB$ は右辺が消去可能変数だけから成る規則なので，S も消去可能である．したがって，3 個の変数 S, A, B はすべて消去可能である．

さて文法 G_1 の規則を構成しよう．まず，$S \to AB$ を考える．この規則の右辺に現れる記号はすべて消去可能である．それゆえ，A, B それぞれのあるなしで，4 通りの可能性がある．しかし，すべての記号を消すわけにはいかないので，

$$S \to AB \mid A \mid B$$

の 3 規則を P_1 に加える．

次に，生成規則 $A \to aAA$ を考えよう．2 番目と 3 番目の位置に，消去可能な変数があるので，再びそれらのあるなしで 4 通りの可能性がある．今度は，消去可能でない終端記号 a が現れているため，4 通りをすべて追加する．

$$A \to aAA \mid aA \mid aA \mid a$$

中の二つはたまたま一致していることに注意せよ．A を一つ消すのであれば，どちらの A を消すかは問題ではない．したがって，文法 G_1 は A に対する規則を三つ持つ．

同様に，B に対する G_1 の規則は，

$$B \to bBB \mid bB \mid b$$

になる．G の二つの ε-規則は G_1 には加えないので，次の規則が G_1 を構成する．

$$S \to AB \mid A \mid B$$

$$A \to aAA \mid aA \mid a$$

$$B \to bBB \mid bB \mid b$$

▨

ε-規則の消去に関する学習の最後に，上記の構成は，ε を除いて，生成言語を変えないことを証明しよう．構成は明らかに ε-規則を除いているので，すべての CFG G に対して，ε-規則を持たない文法 G_1 で

$$L(G_1) = L(G) - \{\varepsilon\}$$

を満たすものが存在することの完全な証明が得られる．

定理 7.9 文法 G_1 が上記の ε-規則の消去で文法 G から構成されるならば，$L(G_1) = L(G) - \{\varepsilon\}$ である．

証明 $w \neq \varepsilon$ ならば，$w \in L(G_1)$ である必要十分条件は $w \in L(G)$ であることを示す．よくあるように，より一般的な主張を証明する方がやさしい．この場合，直接関係する S が生成する記号列だけでなく，各変数 A が生成する記号列について，次のことを証明する．

$$A \overset{*}{\underset{G_1}{\Rightarrow}} w \text{ である必要十分条件は } A \overset{*}{\underset{G}{\Rightarrow}} w \text{ かつ } w \neq \varepsilon \text{ である．}$$

（十分性） $A \overset{*}{\underset{G_1}{\Rightarrow}} w$ とする．G_1 は ε-規則を持たないので，$w \neq \varepsilon$ である．また G_1 の規則は，G の規則から消去可能変数 (の一部) を消して得られる規則なので，任意の変数 B に対し，$B \to \beta$ が G_1 の規則ならば $B \overset{*}{\underset{G}{\Rightarrow}} \beta$ である．したがって，G_1 における導出の 1 ステップは G の数ステップの導出に展開でき，$A \overset{*}{\underset{G_1}{\Rightarrow}} w$ ならば $A \overset{*}{\underset{G}{\Rightarrow}} w$ である．

（必要性） $A \overset{*}{\underset{G}{\Rightarrow}} w$ かつ $w \neq \varepsilon$ とする．導出 $A \overset{*}{\underset{G}{\Rightarrow}} w$ のステップ数 n に関する帰納法で $A \overset{*}{\underset{G_1}{\Rightarrow}} w$ を証明する．

基礎：ステップ数 $n = 1$ のとき，$A \to w$ は G の規則である．$w \neq \varepsilon$ なので，この規則は G_1 の規則でもあり，$A \overset{*}{\underset{G_1}{\Rightarrow}} w$ である．

帰納：導出 $A \overset{*}{\underset{G}{\Rightarrow}} w$ のステップ数が $n > 1$ のとき，導出は $A \underset{G}{\Rightarrow} Y_1 Y_2 \cdots Y_m \overset{*}{\underset{G}{\Rightarrow}} w$ の形をしている．さらに，$w = w_1 w_2 \cdots w_m$ と分割できて，$i = 1, 2, \ldots, m$ に対して $Y_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ が成り立つ．$X_1 X_2 \cdots X_k$ を $w_j \neq \varepsilon$ である Y_j を順に並べて得られる記号列とする．各導出 $Y_j \overset{*}{\underset{G}{\Rightarrow}} w_j$ の長さは n 未満なので，帰納法の仮定より，$w_j \neq \varepsilon$ ならば $Y_j \overset{*}{\underset{G_1}{\Rightarrow}} w_j$ である．したがって $A \underset{G_1}{\Rightarrow} X_1 X_2 \cdots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$ である．

証明は以下のように完成する．上記で $A = S$ とおけば，$S \overset{*}{\underset{G_1}{\Rightarrow}} w$ の必要十分条件は，$S \overset{*}{\underset{G}{\Rightarrow}} w$ かつ $w \neq \varepsilon$ である．すなわち，$w \in L(G_1)$ の必要十分条件は，$w \in L(G)$ かつ $w \neq \varepsilon$ である． ▨

7.1.4　単位規則の除去

単位規則 (unit production) は $A \to B$ (A, B は変数) という形の規則である．たとえば，例 5.27 の単純な数式を生成する文法

$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

$$F \to I \mid (E)$$

$$T \to F \mid T * F$$

$$E \to T \mid E + T$$

では，単位規則 $E \to T$ と $T \to F$ を使ってあいまいでない文法を構成した．しかし，単位規則は証明を複雑にし，技術的には余分なステップを導出にもたらす．たとえば，規則 $E \to T$ の T は 2 通りに展開でき，二つの規則 $E \to F \mid T * F$ に置き換えることができる．この変更は，もとの文法になかった新たな単位規則 $E \to F$ をもたらすので，単位規則を減らすわけではない．そこで $E \to F$ を F の二つの規則を使ってさらに展開すると，$E \to I \mid (E) \mid T * F$ が得られる．まだ，単位規則 $E \to I$ が残る．しかし，I を左辺に持つ六つの規則を使ってさらに展開すれば，$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F \mid E + T$ を得る．

こうして，E の単位規則は消えた．$E \to a$ は，右辺の記号が終端記号であって変数ではないから，単位規則ではない．

上で示した，単位規則をそれが消えるまで展開するという技法でもうまくいくことが多いが，$A \to B, B \to C, C \to B$ といった単位規則のサイクルがあると失敗する．いつでもうまくいく方法は，あらかじめ，単位規則だけを使って $A \overset{*}{\Rightarrow} B$ となる変数 A と B の対を見つけておく方法である．一度そのような対を見つけてしまえば，導出 $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_n \Rightarrow \alpha$ を，A に非単位規則 $B_n \to \alpha$ を直接適用する規則 $A \to \alpha$ に置き換えることができる．ただし，単位規則がなくても，たとえば規則 $A \to BC$ と $C \to \varepsilon$ があれば，$A \overset{*}{\Rightarrow} B$ となることがあることに注意しよう．

単位規則だけを使って $A \overset{*}{\Rightarrow} B$ となる変数の対 (A, B) の帰納的構成を示そう．このような対を**単位対** (unit pair) という．

基礎：(A, A) は単位対である．すなわち，長さ 0 の導出で $A \overset{*}{\Rightarrow} A$ である．

帰納：(A, B) が単位対で $B \to C$ が単位規則ならば，(A, C) も単位対である．

例 7.10 上で再掲した例 5.27 の数式の文法を考えよう．基底ステップで単位対 (E, E), (T, T), (F, F), (I, I) が得られる．帰納ステップでは，次のような構成が行われる．

(1) (E, E) と規則 $E \to T$ から単位対 (E, T) を得る．
(2) (E, T) と規則 $T \to F$ から単位対 (E, F) を得る．
(3) (E, F) と規則 $F \to I$ から単位対 (E, I) を得る．
(4) (T, T) と規則 $T \to F$ から単位対 (T, F) を得る．
(5) (T, F) と規則 $F \to I$ から単位対 (T, I) を得る．
(6) (F, F) と規則 $F \to I$ から単位対 (F, I) を得る．

これ以上構成される単位対はない．そして実際，これら 10 個の対が単位規則だけを使って得られるすべての単位対である． ▨

これ以降の議論の展開はもはやお馴染みだろう．提案されたアルゴリズムが望みの対をすべて得ることにはやさしい帰納的証明がある．次に，その対に関する知識を使って文法から単位規則を除去しても，生成する言語が変わらないことを示す．

定理 7.11 上記のアルゴリズムは CFG G の単位対を正しく計算する．

証明 対 (A, B) が見つかれば，単位規則だけを使って $A \underset{G}{\overset{*}{\Rightarrow}} B$ であることの証明は，対が見つかるまでの手順に関する簡単な帰納法である．この証明は読者に残しておく．

もう一方の証明のために，単位規則だけを使って，$A \underset{G}{\overset{*}{\Rightarrow}} B$ であると仮定する．導出の長さに関する帰納法で，対 (A, B) が見つかることを示す．

基礎：導出の長さが 0 のとき，$A = B$ であり，対 (A, B) はアルゴリズムの基底ステップで構成される．

帰納：$A \overset{*}{\Rightarrow} B$ の導出の長さが $n > 0$ であり，導出の各ステップは単位規則の適用であるとする．このとき，導出は $A \overset{*}{\Rightarrow} C \Rightarrow B$ の形をしている．導出 $A \overset{*}{\Rightarrow} C$ の長さは $n - 1$ なので，帰納法の仮定により，対 (A, C) はアルゴリズムで構成される．したがって，アルゴリズムの帰納ステップで，(A, C) と規則 $C \to B$ が対 (A, B) を構成するのに使われる． ▨

単位規則を除去するには，与えられた CFG $G = (V, T, P, S)$ に対し，次のように CFG $G_1 = (V, T, P_1, S)$ を構成する．

(1) G のすべての単位対を求める．

(2) すべての単位対 (A, B) と G の非単位規則 $B \to \alpha$ とに対し，規則 $A \to \alpha$ を P_1 に追加する．したがって，P_1 の規則はすべて非単位規則である．

例 7.12 例 7.10 では，例 5.27 の数式の文法に対して，アルゴリズムのステップ (1) を実行した．下図 7.1 にステップ (2) をまとめた．ここで，新しい規則は，対の最初の変数を規則の頭部に使い，対の 2 番目の変数を頭部に持つすべての非単位規則の本体を使って，得られる．

対	規則
(E, E)	$E \to E + T$
(E, T)	$E \to T * F$
(E, F)	$E \to (E)$
(E, I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \to T * F$
(T, F)	$T \to (E)$
(T, I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \to (E)$
(F, I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

図 7.1 単位規則除去アルゴリズムのステップ (2) で構成される文法

その結果の文法

$$E \to E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

$$T \to T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

$$F \to (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

は，単位規則を持たず，図 5.19 の文法と同じ数式の集合を生成する． ▨

定理 7.13 文法 G_1 が文法 G から上記の単位規則除去アルゴリズムで構成されるとき，$L(G_1) = L(G)$ である．

証明 $w \in L(G_1)$ の必要十分条件が $w \in L(G)$ であることを示す．

(必要性) G_1 のすべての生成規則は，G の 0 個以上の単位規則を適用した後に非単位規則を適用して得られたものであるから，G_1 における導出の各ステップは，G の長さ 1 以上の導出で置き換えることができる．したがって，$S \overset{*}{\underset{G_1}{\Rightarrow}} w$ ならば，$S \overset{*}{\underset{G}{\Rightarrow}} w$ で

ある．

（十分性） $w \in L(G)$ とする．5.2 節の同値性から，w は最左導出 $S \underset{lm}{\Rightarrow} w$ を持つ．単位規則が最左導出で使われるときは，その本体はまた最左変数になり，直ちに置き換えられる．したがって，G における最左導出は，0 回以上の単位規則の適用に続く非単位規則の適用という導出ブロックに分割される．先行する単位規則を持たない非単位規則は，それ自身で一つの導出ブロックをなす．G_1 の構成は，単位規則に続く非単位規則の適用を反映しているので，これらの導出ブロックはそれぞれ G_1 の一つの規則で実現される．したがって，$S \overset{*}{\underset{G_1}{\Rightarrow}} w$ である． ▨

ここまでで議論した種々の単純化をまとめておこう．CFG G を，無用な記号を持たず，ε-規則や単位規則を持たない等価な CFG に変換した．個々の構成を適用する順番には少し注意が必要である．安全な順番は

(1) ε-規則を除く

(2) 単位規則を除く

(3) 無用な記号を除く

である．7.1.1 項で二つのステップを正しい順番で行わないと，無用な記号が残る心配があったのと同様に，上の三つのステップも上に述べた順番で行わないと，除いたはずの規則が残ることがある．

定理 7.14 G が ε 以外の列を少なくとも一つ生成する文法とするとき，$L(G_1) = L(G) - \{\varepsilon\}$ で，ε-規則や単位規則，無用な記号を持たない文法 G_1 が存在する．

証明 まず，7.1.3 項の方法で ε-規則を除く．次に 7.1.4 項の方法で単位規則を除く．このとき，新しい規則の本体は，古い規則の本体のどれかに一致しているので，ε-規則が導入されることはない．最後に，無用な記号を 7.1.1 項の方法で除く．この変換は規則と記号を除くだけなので，新しい規則を導入することはなく，結果として得られる文法は ε-規則と単位規則を持たない． ▨

7.1.5 チョムスキーの標準形

文法の単純化の学習を，ε を含まない空でない CFL は，文法の規則がすべて次の 2 種類の単純な形のどちらかをした文法を持つことを示すことで締めくくろう．

(1) $A \to BC$，ここで A, B, C は変数である．

(2) $A \to a$，ここで A は変数で a は終端記号である．

さらに，G は無用な記号を含まない．そのような文法を**チョムスキー標準形**，CNF(Chomsky Normal Form) という[1]．

文法を CNF にするために，定理 7.14 の単純化を満たす文法から始めよう．すなわち，文法は ε-規則や単位規則，無用な記号を持たない．そのような文法の規則は，$A \to a$ というすでに CNF で許される形の規則であるか，長さ 2 以上の本体を持つかのどちらかである．課題は，

a) 長さ 2 以上の本体は変数だけから成るようにする．

b) 長さ 3 以上の本体を，長さ 2 の本体を持つ規則の列に分解する．

ステップ (a) は次のように進む．長さ 2 以上の本体に現れる終端記号 a に対し，新しい変数 (A とする) を作る．この変数に，ただ一つの規則 $A \to a$ を加え，長さ 2 以上の本体に現れる a をすべて A に置き換える．各終端記号に対しこの操作を行った時点で，各規則の本体は一つの終端記号から成るか，長さ 2 以上の終端記号を含まない変数列であるかのどちらかである．

ステップ (b) では，$k \geq 3$ に対する規則 $A \to B_1B_2 \cdots B_k$ を，長さ 2 の変数列から成る本体を持つ一連の規則に分けなければならない．$k-2$ 個の新しい変数 $C_1, C_2, \cdots, C_{k-2}$ を導入し，元の規則を $k-1$ 個の規則

$$A \to B_1C_1,\ C_1 \to B_2C_2, \ldots, C_{k-3} \to B_{k-2}C_{k-2},\ C_{k-2} \to B_{k-1}B_k$$

で置き換える．

例 7.15 例 7.12 の文法を CNF に変換しよう．8 個の終端記号 a, b, 0, 1, +, $*$, (,) があり，どれも長さ 2 以上の本体中に現れる．したがって，ステップ (a) では，8 個の新しい変数と，それを対応する終端記号に書き換える 8 個の規則を導入しなければならない．各記号の (英語の) イニシャルを利用することにして，規則

$$\begin{array}{llll} A \to a & B \to b & Z \to 0 & O \to 1 \\ P \to + & M \to * & L \to (& R \to) \end{array}$$

---

[1] チョムスキーは自然言語を記述する道具として最初に文脈自由文法を提案した言語学者である．彼はすべての CFG がこの形に変換できることを示した．面白いことに，CNF は自然言語では重要な用途を持たないようであるが，CFL の所属問題の効率的な検査などのような他の用途があることを後に見る (7.4.4 項)．

を導入し，長さ 2 以上の本体に現れるすべての終端記号を対応する変数で置き換えると，図 7.2 の文法を得る．

$$
\begin{array}{lcl}
E & \rightarrow & EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
T & \rightarrow & TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
F & \rightarrow & LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO \\
I & \rightarrow & a \mid b \mid IA \mid IB \mid IZ \mid IO \\
A & \rightarrow & a \\
B & \rightarrow & b \\
Z & \rightarrow & 0 \\
O & \rightarrow & 1 \\
P & \rightarrow & + \\
M & \rightarrow & * \\
L & \rightarrow & (\\
R & \rightarrow &)
\end{array}
$$

図 7.2　本体をすべて一つの終端記号か長さ 2 以上の変数列にする

ここで，長さ 3 以上の本体，すなわち EPT, TMF, LER を持つ規則を除いて他の規則はすべてチョムスキー標準形になっている．これらの本体の中には複数個所に現れるものもあるが，本体ごとに新しい変数を導入し，各本体を一度だけ処理すればよい．たとえば EPT に対し，新しい変数 C_1 を導入し，それが現れる規則 $E \rightarrow EPT$ を，規則 $E \rightarrow EC_1$ と $C_1 \rightarrow PT$ に置き換える．

TMF に対して新しい変数 C_2 を導入する．この本体を使う二つの規則 $E \rightarrow TMF$ と $F \rightarrow TMF$ を，規則 $E \rightarrow TC_2$, $T \rightarrow TC_2$, $C_2 \rightarrow MF$ に置き換える．LER に対しては新しい変数 C_3 を導入し，それを用いる三つの規則 $E \rightarrow LER$, $T \rightarrow LER$, $F \rightarrow LER$ を，$E \rightarrow LC_3$, $T \rightarrow LC_3$, $F \rightarrow LC_3$, $C_3 \rightarrow ER$ に置き換える．最終的に得られる文法を図 7.3 に示した．▨

定理 7.16　G を ε 以外の列をすくなくとも一つ生成する CFG とする．$L(G_1) = L(G) - \{\varepsilon\}$ を満たすチョムスキー標準形文法 G_1 が存在する．

証明　定理 7.14 より，$L(G_2) = L(G) - \{\varepsilon\}$ となる CFG G_2 で，ε-規則や単位規則，無用な記号を持たないものが見つかる．G_2 を CNF G_1 に変換する構成では，G_2 の規則を G_1 の一つ以上の規則で模倣する．逆に，G_2 で新しく導入された変数は，高々一つの規則しか持たないので，意図したようにしか用いられることはない．以下，

$$
\begin{array}{lcl}
E & \rightarrow & EC_1 \,|\, TC_2 \,|\, LC_3 \,|\, a \,|\, b \,|\, IA \,|\, IB \,|\, IZ \,|\, IO \\
F & \rightarrow & TC_2 \,|\, LC_3 \,|\, a \,|\, b \,|\, IA \,|\, IB \,|\, IZ \,|\, IO \\
T & \rightarrow & LC_3 \,|\, a \,|\, b \,|\, IA \,|\, IB \,|\, IZ \,|\, IO \\
I & \rightarrow & a \,|\, b \,|\, IA \,|\, IB \,|\, IZ \,|\, IO \\
A & \rightarrow & a \\
B & \rightarrow & b \\
Z & \rightarrow & 0 \\
O & \rightarrow & 1 \\
P & \rightarrow & + \\
M & \rightarrow & * \\
L & \rightarrow & (\\
R & \rightarrow &) \\
C_1 & \rightarrow & PT \\
C_2 & \rightarrow & MF \\
C_3 & \rightarrow & ER
\end{array}
$$

図 7.3　本体をすべて一つの終端記号か長さ 2 の変数列にする

$w \in L(G_2)$ が $w \in L(G_1)$ の必要十分条件であることを形式的に証明する．

（十分性）　w が G_2 での導出を持てば，使われている規則，$A \rightarrow X_1X_2 \cdots X_k$ を G_1 での導出列に置き換えるのはやさしい．すなわち，G_2 での 1 回の導出は，G_1 の規則を用いた 1 回以上の導出ステップに置き換えられる．まず，X_i が終端記号の場合，対応する変数 B_i と規則 $B_i \rightarrow X_i$ が G_1 にある．さらに，$k > 2$ ならば，G_1 は規則 $A \rightarrow B_1C_1, C_1 \rightarrow B_2C_2, \ldots$ を持つ．ここで，B_i は終端記号 X_i に対して導入された変数であるかまたは変数 X_i それ自身である．これらの規則は G_2 で $A \rightarrow X_1X_2 \cdots X_k$ を使った導出 1 ステップを模倣する．したがって w は G_1 での導出を持ち，$w \in L(G_1)$ である．

（必要性）　$w \in L(G_1)$ とする．S を根とし w を産物とする G の構文木がある．この木を，やはり S を根とし w を産物とする G_2 の構文木に変換する．

最初に，CNF の構成のステップ (b) を “元へ戻す”．A をラベルに持つ節点がラベル B_1 と C_1 の二つの子を持つとしよう．ここで C_1 は (b) で新しく導入された変数である．このとき，構文木のこの部分は，図 7.4(a) のようになる．すなわち，規則 $A \rightarrow B_1B_2 \cdots B_k$ を処理するために新しく導入されたすべての変数は，規則を一つしか持たず，構文木中への現れ方は 1 通りで，図に示したように連続して現れなければならない．

構文木におけるこのような固まり (部分木) は，それが表す G_2 の一つの規則に置き換えることができる．構文木の変換を図 7.4(b) に示した．

その結果得られる構文木はまだ G_2 の構文木ではない．その理由は，CNF を構成するステップ (a) で，一つの終端記号を導出する新たな変数を導入しているからである．

しかしながら，現時点の構文木中で，対応する部分木を取り出し，変数 A でラベル付けされた節点とラベル a を持つその子とを，ラベル a の一つの節点で置き換えることは容易にできる．この結果，構文木のすべての内節点は G_2 の規則を表すようになる．w は G_2 の構文木の産物であるので，$w \in L(G_2)$ である．▨

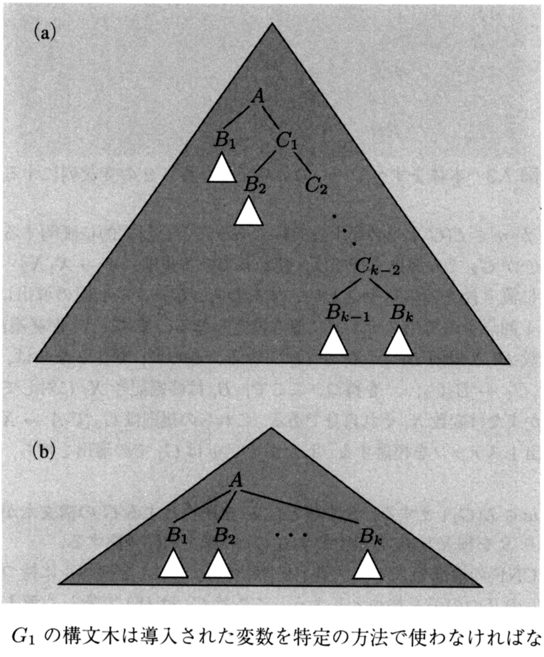


図 7.4　G_1 の構文木は導入された変数を特定の方法で使わなければならない

グライバッハ標準形

証明はしないが，CFG の別の興味ある標準形が知られている．すべての ε を持たない非空 CFL は，その規則がすべて $A \to a\alpha$ (a は終端記号で α は 0 個以上の変数の列) の形をした CFG で生成される．文法をこの形に変換するのは，チョムスキー標準形から始めて，問題を単純化させたとしても複雑である．大雑把にいって，各規則の本体の最初の変数を，頭に終端記号が得られるまで展開する．しかし，展開過程が循環して終端記号に到達できないことがあり，その場合，展開を“短絡”させて，本体の頭に終端記号を持ち，それに続く変数から展開過程で生じる変数列をすべて生成できる，そのような規則を作る必要がある．

最初にその構成法を示したグライバッハ (S. Greibach) に因んで，**グライバッハ標準形** (Greibach Normal Form) と呼ばれるこの形からいくつかの興味ある事実が導かれる．規則の適用はいつも 1 個の終端記号を文形式に導入するので，長さ n の記号列は，ちょうど長さ n の導出を持つ．同様に，定理 6.13 の PDA 構成法をグライバッハ標準形文法に適用すれば，ε-遷移を持たない PDA を得る．しがって，PDA から ε-遷移を除くことは常に可能である．

7.1.6 7.1 節の演習問題

♣ ***問 7.1.1** 次の文法と等価で無用な記号を持たない文法を求めよ．

$$S \to AB \mid CA$$
$$A \to a$$
$$B \to BC \mid AB$$
$$C \to aB \mid b$$

♣ ***問 7.1.2** 文法

$$S \to ASB \mid \varepsilon$$
$$A \to aAS \mid a$$
$$B \to SbS \mid A \mid bb$$

に対して，

a) ε-規則を除去せよ．
b) 単位規則を除去せよ．
c) 無用な記号があるか．あれば除去せよ．
d) チョムスキー標準形に変換せよ．

♣ **問 7.1.3** 問 7.1.2 を文法

$$S \to 0A0 \mid 1B1 \mid BB$$
$$A \to C$$
$$B \to S \mid A$$
$$C \to S \mid \varepsilon$$

に対して繰り返せ.

♣ **問 7.1.4**　問 7.1.2 を文法

$$S \to AAA \mid B$$
$$A \to aA \mid B$$
$$B \to \varepsilon$$

に対して繰り返せ.

♣ **問 7.1.5**　問 7.1.2 を文法

$$S \to aAa \mid bBb \mid \varepsilon$$
$$A \to C \mid a$$
$$B \to C \mid b$$
$$C \to CDE \mid \varepsilon$$
$$D \to A \mid B \mid ab$$

に対して繰り返せ.

♣ **問 7.1.6**　対応のとれた括弧の列の集合を生成する CNF 文法を書け. 直接 CNF 文法を構成してもよい.

♣!! **問 7.1.7**　文法 G は p 個の規則を持ち, 規則の本体はすべて長さ n 以下とする. $A \overset{*}{\underset{G}{\Rightarrow}} \varepsilon$ ならば, A から ε を導出する長さ $(n^p - 1)/(n - 1)$ ステップ以下の導出があることを示せ.

♣ ! **問 7.1.8**　G は規則の右辺の長さの総和が n で ε 規則を持たないとする. G を CNF に変形するとき,

a) CNF 文法は高々 $O(n^2)$ 個の規則しか持たないことを示せ.

b) CNF 文法が n^2 に比例する個数の規則を持つような文法があり得ることを示せ. **ヒント**: 単位規則を除去する方法について考えよ.

♣ **問 7.1.9**　次の定理の証明を完成させるのに必要な帰納法による証明を与えよ.

a) 定理 7.4 での, 加えた記号が本当に生成的であることの証明

b) 定理 7.6 での, 7.1.2 項の可到達記号を求めるアルゴリズムが正しいことの双方向の証明

c) 定理 7.11 での, 構成した対が本当に単位対であることの証明

♣*! **問 7.1.10**　ε を含まない任意の文脈自由言語に対して，それを生成する文法で規則の形が $A \to BCD$ (すなわち，本体は三つの変数から成る) か，$A \to a$ (本体は終端記号一つ) に限るものは可能か．証明か反例を与えよ．

♣ **問 7.1.11**　ε 以外に少なくとも一つの列を含む文脈自由言語 L に対して，$L-\{\varepsilon\}$ を生成するグライバッハ標準形の CFG が存在することを示そう．グライバッハ標準形 (GNF) 文法は，すべての規則の本体が終端記号で始まるものであった．構成は，一連の構成と補題を使ってなされる．

a)　CFG G が規則 $A \to \alpha B\beta$ を持ち，B の規則が $B \to \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n$ であるとする．このとき，規則 $A \to \alpha B\beta$ の本体の B を B-規則で展開して得られる規則群 $A \to \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \cdots \mid \alpha\gamma_n\beta$ で規則 $A \to \alpha B\beta$ を置き換えたとき，得られる文法はもとの文法と同じ言語を生成することを示せ．

以下，L の文法 G は，チョムスキー標準形であり，変数は $A_1, A_2, \cdots, A_k$ であるとする．

*!b)　(a) の変換を繰り返し適用することにより，G と等価で，A_i のすべての規則が，終端記号かある A_j $(j \geq i)$ かで始まるものが得られることを示せ．どちらの場合も，本体の最初の記号より後はすべて変数である．

!c)　ステップ (b) を G に適用した結果を G_1 とする．A_i を変数，$A_i \to A_i\alpha_1 \mid A_i\alpha_2 \mid \cdots \mid A_i\alpha_m$ を本体が A_i で始まる A_i-規則の全体であるとし，

$$A_i \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_p$$

をそれ以外の A_i-規則の全体であるとする．各 β_j は終端記号か，j より大きな添え字を持つ変数で始まることに注意せよ．新しい変数 B_i を導入して，最初の m 個の規則のグループを，

$$A_i \to \beta_1 B_i \mid \cdots \mid \beta_p B_i$$
$$B_i \to \alpha_1 B_i \mid \alpha_1 \mid \cdots \mid \alpha_m B_i \mid \alpha_m$$

に置き換える．その結果得られる文法は G や G_1 と同じ言語を生成することを示せ．

*!d)　G_2 をステップ (c) で得られた文法とする．A_i-規則の本体はすべて，終端記号か A_j $(j > i)$ で始まる．さらに，B_i-規則の本体もすべて，終端記号か A_j $(j > i)$ で始まる．G_2 は等価な GNF の文法を持つことを示せ．**ヒント**: 最初に，ステップ (a) を使って A_k-規則を定めよ．次に，A_{k-1}-規則を定め，以下同様にして A_1-規則まで定める．次に，B_i-規則を任意の順番で，再び (a) を使って，定める．

♣ **問 7.1.12**　問 7.1.11 の構成を使って，文法

$$S \to AA \mid 0$$

$$A \to SS \mid 1$$

を GNF に変換せよ．

7.2 文脈自由言語の反復補題

この節では言語が文脈自由言語でないことを示す定理を開発する．定理は，「文脈自由文法の反復補題」といい，文脈自由言語に属する十分に長い列では，高々二つの互いに近い位置にある短い部分列で，連係して反復できるものが必ず見つかるというものである．すなわち，二つ (以下) の部分列をともに任意の回数繰り返すことができ，得られた列は再びその言語に属する．

この定理は正則言語に対する同様の反復補題と比べるとおもしろい．その定理 4.1 は，反復できる一つの部分列が必ず見つかるというものだった．違いは，$L = \{0^n1^n \mid n \geq 1\}$ のような言語を考えると明らかになる．十分大きな n を定めて，列 0^n1^n 中の 0 の部分列を反復することによって，1 の数より 0 の数のほうが多い列が得られるため，L は正則でないことが示せる．しかし，CFL の反復補題では，二つの部分列が見つかるということがいえるだけである．したがって，0 の列と 1 の列を同時に使うように強制されるかも知れず，反復しても L の列しか得られないかも知れない．実際，L は CFL なので，CFL の反復補題を用いて L に属さない列を作り出すことがいえるはずはない．

7.2.1 構文木の大きさ

CFL の反復補題を導く最初のステップは，構文木の大きさと形を調べることである．CNF の使い道の一つは，構文木が 2 分木になることである．2 分木は便利な性質を持つ．その一つをここで利用しよう．

定理 7.17　チョムスキー標準形の文法 $G = (V, T, P, S)$ の構文木があり，その産物が終端記号列 w であるとする．木の最長路の長さが n ならば，$|w| \leq 2^{n-1}$ である．

証明　証明は n についての単純な帰納法による．

基礎：$n = 1$ とする．路の長さは辺で数えるので，路の節点の個数より 1 少ない．したがって，最長路の長さが 1 であるような木は，根と終端記号でラベル付けされた一つの葉だけから成る．列 w はこの終端記号そのものであり，$|w| = 1$ である．この場

合，$2^{n-1} = 2^0 = 1$ であり，基底ステップは証明された．

帰納：最長路が長さ n $(n > 1)$ であるとする．$n > 1$ なので，構文木の根が使う規則は $A \to BC$ の形をしている．B, C を根とする部分木の路は，もとの木の根からその子 B や C への辺を除いたものなので，長さが $n-1$ を越えることはない．したがって帰納法の仮定により，これら二つの部分木の産物はそれぞれ，長さ 2^{n-2} 以下である．全体の木の産物はこれら木の産物の連接であるから，その長さは $2^{n-2} + 2^{n-2} = 2^{n-1}$ 以下である．よって，帰納ステップも証明された．▨

7.2.2　反復補題の表現

CFL の反復補題は，正則言語の反復補題とよく似ている．しかし，CFL L の列 z は五つの部分列に分割され，2 番目と 4 番目の部分列が反復可能になる．

定理 7.18　**(文脈自由言語の反復補題)** L を CFL とする．ある定数 n が存在して，z が長さ n 以上の L の列ならば，$z = uvwxy$ と書けて，以下の条件が成り立つ．

(1)　$|vwx| \le n$．すなわち，中央部分は余り長くない．

(2)　$vx \ne \varepsilon$．v と x は反復部分であり，反復部分の少なくとも一方は空でない．

(3)　任意の i に対して，$uv^iwx^iy \in L$．すなわち，二つの列 v と x は，連動して 0 回以上の任意の回数反復でき，得られた列は再び L に属する．

証明　まず L のチョムスキー標準形文法 G を構成する．技術的理由から，CFL L が $\emptyset$ や $\{\varepsilon\}$ のときには，そのような文法を構成することはできない．しかし，$L = \emptyset$ なら，L の元 z について主張するこの定理は，もともとそのような z が存在しないことから，反例を持たない．また，CNF 文法 G は実際には $L - \{\varepsilon\}$ を生成するのであるが，このことも再び重要ではない．実際，定理で $n > 0$ が選ばれれば，z が ε となることはどちらにしろあり得ない．

さて，$L(G) = L - \{\varepsilon\}$ なる CNF G が m 個の変数を持つとする．$n = 2^m$ を選び，$z \in L$ の長さが n 以上だとする．定理 7.17 より，最長路の長さが m 以下の構文木は，産物の長さが $2^{m-1} = n/2$ 以下であるので，z を産物に持つことができない．それゆえ，z を産物に持つ構文木は必ず長さ $m+1$ 以上の路を持つ．

図 7.5 は，z の構文木の最長路を示している．最長路の長さは $k+1$ で，$k \ge m$ である．$k \ge m$ なので，路上に少なくとも $m+1$ 個の変数の出現 $A_0, A_1, \ldots, A_k$ がある．V には，m 個の異なる変数しかないので，路上の A_{k-m} から A_k の $m+1$ 個の変数の中の少なくとも二つは同じ変数でなければならない．$A_i = A_j$ $(k - m \le i < j \le k)$

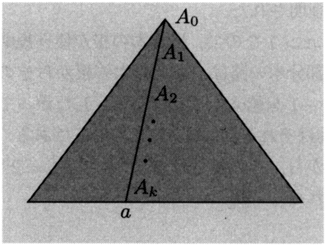


図 7.5　L の十分に長い列の構文木は，長い路を持つ

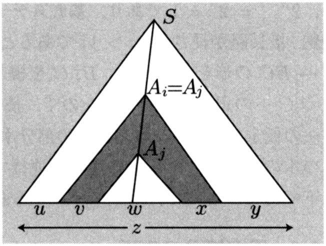


図 7.6　列 w を反復可能な形に分割する

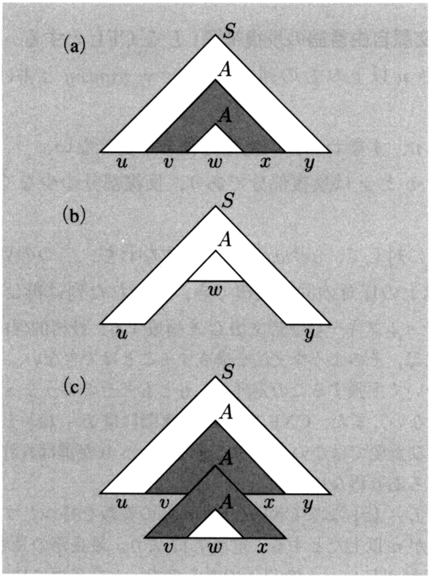


図 7.7　列 v と x の 0 回の反復 (b) および 2 回の反復 (c)

としよう．

このとき，構文木を図 7.6 に示したように分割できる．列 w は A_j を根とする部分木の産物である．列 v と x は A_i を根とするより大きな部分木の産物の，それぞれ w の左側と右側の部分列である．単位規則がないことから，v と x は，一方が ε になる

ことはあっても，ともに ε になることはない．最後に，u と y は A_i を根とする部分木の左右に位置する z の部分列である．

$A_i = A_j = A$ とおくと，元の構文木から，新しい構文木を図 7.7(a) のように，構成できる．最初に，産物 vwx を持ち A_i を根とする部分木を，産物 w を持ち A_j を根とする部分木で置き換える．これができる理由は，それらがともに A を根とするからである．その結果得られる構文木を，図 7.7(b) に示した．その木は産物 uwy を持ち，列のパターン uv^iwx^iy の $i = 0$ の場合にあたる．

別の選択を図 7.7(c) に示した．A_j を根とする部分木を，A_i を根とする部分木全体で置き換えることもできる．再び，その正しさは，根のラベルが A の木を同じ根のラベルを持つ木で置き換えているという点にある．この木の産物は，uv^2wx^2y である．さらに，図 7.7(c) の産物 w を持つ部分木を，産物 vwx を持つより大きな部分木で置き換えれば，産物 uv^3wx^3y を持つ構文木が得られる．これを繰り返して，任意の指数 i に対して，uv^iwx^iy の G の構文木が得られ，反復補題の大部分が証明された．

残っているのは，条件 (1) の $|vwx| \leq n$ である．しかし，A_i は構文木の葉に近いほうから選ばれており，$k - i \leq m$ を満たす．したがって，A_i を根とする部分木の最長路は長さ $m + 1$ 以下で，定理 7.17 より，A_i を根とする部分木の産物 vwx は，その長さが $2^m = n$ を越えない．▨

7.2.3　反復補題の CFL への応用

正則言語に対する以前の反復補題のように，CFL の反復補題を以下のような "敵対ゲーム" として使うことができる．

(1)　CFL でないことを示したい言語 L を選ぶ．

(2)　"敵" は，n を選ぶ．n は我々には決められない任意の数なので，すべての n に対応できるようにしなければならない．

(3)　我々は L に属する長さ n 以上の列 z を選ぶ．

(4)　"敵" は z を $uvwxy$ の形に分割する．分割の条件は $|vwx| \leq n$ と $vx \neq \varepsilon$ だけである．

(5)　ある i を選んで uv^iwx^iy が L に属さないことをいつでも示せれば，"勝" である (L は CFL でない)．

反復補題を使って CFL でないことが示せる言語の例をいくつか示す．最初の例は，CFL では二つの記号グループの一致・不一致を検査することはできるが，三つのグループの検査はできないことを意味する．

例 7.19　$L = \{0^n1^n2^n \mid n \geq 0\}$ とする．すなわち，L は $\mathbf{0^+1^+2^+}$ に属し各記号が同数の列 (012，001122 等々) から成る．L が文脈自由言語だったとしよう．このとき反復補題から与えられる n が存在する[2)]．$z = 0^n1^n2^n$ を選ぶ．

敵が z を $uvwxy$ に分割し，$|vwx| \leq n$ かつ v と x のどちらかは ε でないとする．このとき，z の最後の 0 と最初の 2 の間は $n+1$ 離れているのだから，vwx が記号 0 と 2 をともに含むことはできない．L は L に属さないことがわかっている列を含んでしまい，L が CFL であるという仮定が矛盾することを示そう．場合わけは次のようになる．

(1)　vwx が 2 を含まない場合．vx は 0 と 1 しか含まず，しかも少なくとも一つは含む．すると反復補題から L に属すことになっている uwy は，n 個の 2 と n 個未満の 0 か 1 を含むことなる．それゆえ，uwy は L に属さず，この場合 L は CFL でないと結論できる．

(2)　vwx が 0 を含まない場合．同様に，uwy は n 個の 0 を含み，n 個未満の 1 か 2 を含む．それゆえ uwy は L に属さない．

どちらの場合も，L は L に属さないはずの列を含むことになる．この矛盾から，仮定が間違っていたことがわかり，L は CFL でない．　▨

CFL にできない別のことは，同数の記号の対二つを交互に挟むことである．そのアイデアは反復補題を使った非文脈自由言語性の証明の次の例で明確になる．

例 7.20　$L = \{0^i1^j2^i3^j \mid i \geq 0, j \geq 0\}$ とする．L が文脈自由言語ならば，n を L の定数とし，$z = 0^n1^n2^n3^n$ を選ぶ．$z = uvwxy$ と書けて，$|vwx| \leq n$ かつ $vx \neq \varepsilon$ を満たす．このとき，vwx は一つの記号だけから成る部分列であるか，または二つの隣あう記号にまたがる部分列かのどちらかである．

vwx が一つの記号だけから成る場合，uwy は n 個の 3 種類の記号と，残りの n 個未満の記号から成る列であり，L に属さない．vwx が二つの (z 上で隣り合う) 記号，1 と 2 としよう，から構成される場合，uwy ではいくつかの 1 または 2 (あるいはその両方) が欠けることになる．1 が欠けているとすると 3 は n 個あるので，uwy は L に属さない．同様に，2 が欠けている場合も 0 は n

2)　この n は反復補題が定める定数であり，言語 L 自身の定義に現れた局所変数 n とは無関係である．

個あるので，uwy は L に属さない．結局，L が CFL であるという仮定は矛盾するので，CFL ではない．▨

最後の例として，CFL は，2 個以上の記号を持つアルファベット上の任意の長さの二つの列を一致させることはできないことを示そう．ちなみにこの結果の意味は，文脈自由文法は，プログラミング言語に対して——名前は使われる前に宣言されなければならいといった——ある種の意味的制限を強制するのに適した機構ではない，ということである．現実的には,「記号表」といった別の機構が宣言された名前を記録するのに使われ，構文解析部がそれ自身で「使用前の宣言」を検査するように設計されたりはしない．

例 7.21　$L = \{ww \mid w \in \{0,1\}^*\}$ とする．L は，$\varepsilon, 0101, 00100010, 110110$ などといった列の繰り返しから成る．L を文脈自由言語とし，n を反復補題の定数とする．列 $z = 0^n1^n0^n1^n$ を考えよう．これは 0^n1^n の繰り返しなので，L に属する．

以前の例にならって，$|vwx| \leq n$ かつ $vx \neq \varepsilon$ となるように $z = uvwxy$ と分割する．uwy が L に属さないことを示して矛盾を導き，L が CFL でないことを示す．

まず，$|vwx| \leq n$ なので，$|uwy| \geq 3n$ である．したがって，uwy が繰り返し列 tt ならば，t の長さは $3n/2$ 以上である．vwx が z のどこに位置するかによって場合分けする．

(1)　vwx が最初の 0^n に含まれる場合：$vx = 0^k$ $(0 < k \leq n)$ とする．このとき $uwy = 0^{n-k}1^n0^n1^n$ である．$|uwy| = 4n - k$ なので，$uwy = tt$ ならば $|t| = 2n - k/2$ である．したがって，t が最初の 1 のブロックの途中で終わることはなく，t は記号 0 で終わる．しかし，uwy は 1 で終わるので，tt と等しいことはあり得ない．

(2)　vwx が最初の 0 のブロックと最初の 1 のブロックにまたがる場合：この場合でも，$x = \varepsilon$ なら $vx = 0^k$ となることはあり得る．そのときは (1) と同じ議論で uwy が tt の形にならないことが示せる．vx が 1 個でも 1 を含めば，$3n/2$ 以上の長さを持つ t は，uwy が 1^n で終わっていることから，1^n で終わらなければならない．しかし，uwy 中の 1^n のブロック

は，最後のブロック以外にはないので，t を uwy 中で繰り返すことはできない．

(3)　vwx が 1^n の最初のブロックに含まれる場合：uwy が L に属さないことの議論は (2) の後半と同様である．

(4)　vwx が最初の 1^n と 2 番目の 0^n にまたがる場合：vx が実際には 0 を含まないときは，(3) の vwx が最初の 1 のブロックに含まれるときと議論は同じである．vx が少なくとも 1 個 0 を含めば，uwy は 0^n で始まり，$tt = uwy$ を満たす t も 0^n で始まる．しかし，二つ目の t に対する 0^n は uwy には存在しない．この場合もまた，uwy は L に属さない．

(5)　他の場合：vwx は z の後半に含まれ，議論は vwx が z の前半に含まれる場合と対称になる．

したがって，どの場合も uwy は L に属さず，L は文脈自由言語でない．　▨

7.2.4　7.2 節の演習問題

♣　**問 7.2.1**　CFL の反復補題を用いて，次の言語が文脈自由でないことを示せ．

*****a)　$\{a^i b^j c^k \mid i < j < k\}$.

b)　$\{a^n b^n c^i \mid i \le n\}$.

c)　$\{0^p \mid p$ は素数$\}$.

ヒント：正則言語でないことを示した例 4.3 と同様のアイデアを用いよ．

***!**d)　$\{0^i 1^j \mid j = i^2\}$.

!e)　$\{a^n b^n c^i \mid n \le i \le 2n\}$.

!f)　$\{ww^R w \mid w \in \{0,1\}^*\}$. すなわち，001100001 のようにある列 w の次にその逆順 w^R が続き，再び w が続いているような列の全体．

♣ **! 問 7.2.2**　反復補題を CFL に適用すると，“敵が勝ち”，(非 CFL の) 証明は完成できない．L として次の言語を選んだとき，どこがうまくいかないかを示せ．

a)　$\{00, 11\}$.

*****b)　$\{0^n 1^n \mid n \ge 1\}$.

*****c)　アルファベット $\{0,1\}$ 上の回文の全体 $\{w \mid w \in \{0,1\}^*$ かつ $w = w^R\}$.

♣ **!問 7.2.3**　CFL の反復補題より強い形のオグデン (Ogden) の補題が知られている．反復補題との違いは z の任意に指定した異なる n 個の位置に焦点をあてて，反復される部分列が 1 から n までの指定された位置のどれかを含むことが保証されるという点である．この補題の能力は，言語が二つの部分から成る列を含み，その一方は，反復しても言語から外れる列にならないが，もう一方を反復すると言語に属さない列が得られるという場合に発揮される．この場合，反復が後者の

部分で起きることを示さずに，非 CFL 性の証明を完成させることはできない．オグデンの補題の形式的な表現：L が CFL ならば，ある定数 n が存在して，z が長さ n 以上の列で，n 個以上の z 中の位置を任意に指定しとき，$z = uvwxy$ と書けて，以下が成り立つ．

(1)　vwx に含まれる指定された位置は n 個以下．

(2)　vx は少なくとも 1 個の指定された位置を含む．

(3)　任意の i に対して，$uv^iwx^iy \in L$.

オグデンの補題を証明せよ．

ヒント：証明は定理 7.18 の反復補題の証明と本質的に同じである．指定されなかった位置はもともとなかったかのように考えて，z の構文木の最長路を選ぶ．

♣ * **問 7.2.4**　オグデンの補題 (問 7.2.3) を使って，例 7.21 の $\{ww \mid w \in \{0,1\}^*\}$ が CFL でないことの証明を簡単化せよ．

♣ **問 7.2.5**　オグデンの補題 (問 7.2.3) を使い，次の言語が CFL でないことを証明せよ．

!a)　$\{0^i1^j0^k \mid j = \max(i,k)\}$.

!!b)　$\{a^nb^nc^i \mid i \neq n\}$. **ヒント**：$n$ をオグデンの補題の定数とし，列 $a^nb^nc^{n+n!}$ を考えよ．

7.3　文脈自由言語の閉包性

本節では，文脈自由言語 (CFL) に対する演算で CFL を作り出すことが保証されているものを調べる．閉包性の多くは 4.2 節で正則言語に対して得られたものと類似しているが，違いもある．

まず，代入演算を導入する．代入では，ある言語の列中の記号を，別の言語全体で置き換える．代入は 4.2.3 項で調べた準同型写像の一般化であり，たとえば正則演算 (集合和，連接，閉包) のような他の演算での閉包性の証明に役立つ．さらに CFL が準同型写像や逆順同型写像で閉じていることも示す．正則言語と異なり，CFL は共通部分や集合差で閉じていないが，CFL の正則言語による共通部分や集合差は再び CFL である．

7.3.1　代　　入

Σ をアルファベットとし，Σ の各記号 a に対し，言語 L_a を対応させる．L_a は，必ずしも Σ 上の言語である必要はなく，任意のアルファベット上の言語でよい．この対応が Σ からの写像 s (**代入**：substitution) を定義し，各記号 a に

対し，L_a を $s(a)$ と書く．

$w = a_1a_2\cdots a_n$ が Σ 上の列のとき，$s(w)$ を言語の連接 $s(a_1)s(a_2)\cdots s(a_n)$ で定義する．すなわち，$s(a_1a_2\cdots a_n) = \{x_1x_2\cdots x_n \mid x_1 \in s(a_1), x_2 \in s(a_2), \ldots, x_n \in s(a_n)\}$ である．さらに s の定義を拡張して，言語に適用する．$s(L)$ はすべての $w \in L$ についての $s(w)$ の和 $s(L) = \bigcup_{w\in L} s(w)$ である．

例 7.22 $s(0) = \{a^nb^n \mid n \geq 1\}$，$s(1) = \{aa, bb\}$ とする．すなわち，s は $\Sigma = \{0,1\}$ 上の代入で，言語 $s(0)$ は一つ以上の a に続いて同数の b が続く列から成る言語であり，$s(1)$ は二つの列 aa, bb から成る有限言語である．

$w = 01$ とする．$s(w)$ は言語 $s(0)s(1)$ であるから，a^nb^naa と a^nb^{n+2} $(n \geq 1)$ の形をした列の全体である．

さて，$L = L(\mathbf{0}^*)$，すなわち，0 の任意の有限列全体とする．このとき $s(L) = s(0)^*$ で，

$$a^{n_1}b^{n_1}a^{n_2}b^{n_2}\cdots a^{n_k}b^{n_k}$$

の形をした列の全体である．ただし，$k \geq 0$ かつ $n_1, n_2, \ldots, n_k$ は正整数である．$s(L)$ はたとえば $\varepsilon, aabbaaabbb, abaabbabab$ といった列を含む． ▨

定理 7.23 L を Σ 上の CFL とし，s を Σ 上の代入で各 $a \in \Sigma$ に対し，$s(a)$ が CFL であるとする．このとき，$s(L)$ は CFL である．

証明 本質的なアイデアは，L を生成する CFG 上で，終端記号 a を，$s(a)$ を生成する CFG の開始記号で置き換えることである．その結果，$s(L)$ を生成する CFG が得られる．しかし，このアイデアを実現するためにはうまく定めなければならない細部が数箇所ある．

より形式的には，関連する言語の文法の記述から始める．L の CFG を $G = (V, \Sigma, P, S)$，$s(a)$ $(a \in \Sigma)$ の CFG をそれぞれ $G_a = (V_a, \Sigma_a, P_a, S_a)$ とする．変数には好きな名前を任意に使えるので，これらの文法の変数集合はすべて互いに素だとしてよい．すなわち，二つ以上の V や V_a に共通に含まれる記号 A は存在しない．このように変数名を選ぶ目的は，これらの文法を一つにまとめたとき，二つの文法の規則が偶発的に混合されて，もともとの文法にはなかった導出が作り出されてしまうことを避けるためである．

$s(L)$ の新しい文法 $G' = (V', T', P', S')$ を次のように作る．

- V' は V とすべての V_a $(a \in \Sigma)$ の和．
- T' はすべての T_a $(a \in \Sigma)$ の和．

- P' は次の規則から成る.

(1) P_a $(a \in \Sigma)$ に属するすべての規則.

(2) P の規則で，本体の各終端記号 a をすべて変数 S_a で置き換えたもの.

したがって，文法 G' の構文木は G の構文木のように始まるが，Σ^* に属す産物の代わりに，S_a $(a \in \Sigma)$ をラベルとする節点から成る帯を持つ．さらに節点 S_a から G_a の構文木がぶらさがり，言語 $s(a)$ に属す終端記号列を産物に持つ．G' の典型的な構文木を図 7.8 に示す.

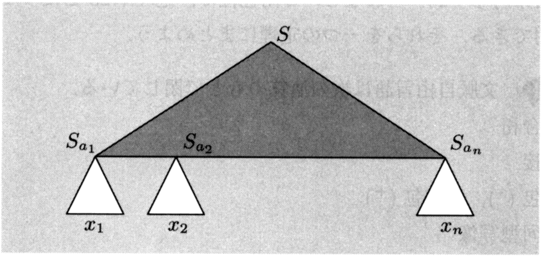


図 7.8 G' の構文木は G の構文木から始まり，それぞれがどれかの G_a に属する多くの構文木で終わる.

さて，G' が $s(L)$ を生成するという意味で，この構成がうまくいっていることを証明しよう.

- 列 $w \in L(G)$ は $w \in s(L)$ の必要十分条件である.

（必要性） $w \in s(L)$ とする．$x = a_1 a_2 \cdots a_n \in L$ と $x_i \in s(a_i)$ $(i = 1, 2, \ldots, n)$ が存在して，$w = x_1 x_2 \cdots x_n$ と書ける．G' のうち，G の生成規則から a を S_a で置き換えて作られた部分は，x 中の a を S_a で置き換えた列 $S_{a_1} S_{a_2} \cdots S_{a_n}$ を導出する．ここまでの導出は図 7.8 の上部の三角形にあたる.

各 G_a の規則は G' の規則でもあり，G_a における S_{a_i} から x_i の導出はそのまま G' の導出でもある．これらの導出の構文木は図 7.8 の下の三角形にあたる．こうして構成された G' の構文木の産物は $x_1 x_2 \cdots x_n = w$ なので，$w \in L(G)$ である.

（十分性） $w \in L(G)$ とする．w の構文木は図 7.8 にようになっている．その理由は，G の変数集合 V と G_a $(a \in \Sigma)$ の変数集合 V_a はすべて互いに素だという点にある．したがって，変数 S から始まる構文木の上部では，S_a が導出されるまで，G の規則だけが使われる．そして S_a より下の部分では，G_a の規則だけが使われる．その結果，w が構文木 T を持つときは常に，以下の条件を満たす $L(G)$ の列 $a_1 a_2 \cdots a_n$ と言語 $s(a_i)$ の列 x_i を特定できる.

(1) $w = x_1x_2\cdots x_n$，かつ

(2) 列 $S_{a_1}S_{a_2}\cdots S_{a_n}$ は T から (図 7.8) に示した下半分の部分木を取り除いた木の産物である．

列 $x_1x_2\cdots x_n$ は，a_i を x_i で置き換えて得られることから，$s(L)$ に属する．したがって，$w \in s(L)$ である． ▨

7.3.2 代入定理の応用

正則言語に対して調べたおなじみの閉包性は，定理 7.23 を使って CFL に対しても証明できる．それらを一つの定理にまとめよう．

定理 7.24 文脈自由言語は次の演算のもとで閉じている．

(1) 集合和

(2) 連接

(3) 閉包 (*)，正閉包 ($^+$)

(4) 準同型写像

証明 それぞれ対応する代入を定義する．以下の証明はすべて CFL への CFL の代入から成り，定理 7.23 より CFL が生じる．以下 $s(1) = L_1, s(2) = L_2$ は CFL であるとする．

(1) 集合和：$L_1 \cup L_2 = s(\{1, 2\})$.

(2) 連接：$L_1L_2 = s(\{12\})$.

(3) 閉包と正閉包：$s(\{1^*\}) = L_1^*$．同様に，$s(\{1^+\}) = L_1^+$ である．

(4) L をアルファベット Σ 上の CFL，h を Σ 上の準同型写像とする．s を各 $a \in \Sigma$ を一つの列 $h(a)$ だけから成る言語に置き換える代入とする．すなわち，すべての $a \in \Sigma$ に対し $s(a) = \{h(a)\}$ とする．このとき，$h(L) = s(L)$ である． ▨

7.3.3 逆　　順

CFL は逆順演算に関しても閉じている．この証明に代入演算を用いることはできないが，文法を利用した簡単な構成法がある．

定理 7.25 L が CFL ならば，L^R も CFL である．

証明 L を生成する CFG を $G = (V, T, P, S)$ から $G^R = (V, T, P^R, S)$ を構成する．ただし，P^R は，P に属する規則を逆順にした規則の集合である．すなわち，$A \to \alpha$ が G の規則ならば，$A \to \alpha^R$ が G^R の規則であるとする．$L(G^R) = L^R$ を示すのは，G や G^R における導出の長さに関する簡単な帰納法である．本質的に，G^R の文形式はすべて G の文形式の逆順になっており，その逆も成り立つ．形式的な証明は演習問

題としよう．　▨

7.3.4　正則言語との共通部分

CFL は共通部分に関して閉じていない．簡単な反例を示そう．

例 7.26　例 7.19 で言語

$$L = \{0^n1^n2^n \mid n \geq 1\}$$

は CFL でないことを学んだ．しかし次の二つの言語は CFL である．

$$L_1 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$$

$$L_2 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$$

L_1 を生成する文法は

$$S \to AB$$

$$A \to 0A1 \mid 01$$

$$B \to 2B \mid 2$$

である．この文法では A から 0^n1^n が生成され，B から 2^i が生成される．L_2 の文法は

$$S \to AB$$

$$A \to 0A \mid 0$$

$$B \to 1B2 \mid 12$$

である．この文法は上の文法と同じように働く．ただし，A からは 0 の任意の列が生成され，B からは 1 と 2 の長さが同じ列が生成される．

しかし，$L = L_1 \cap L_2$ である．列 $w \in L_1 \cap L_2$ は，L_1 に属すことから 0 と 1 の数が等しいことが要請され，L_2 に属すことから 1 と 2 の数が等しいことが要請される．したがって，w では 0, 1, 2 の数がすべて等しくなければならず，$w \in L$ と同値になる．

CFL が共通部分で閉じているとすれば，L が文脈自由言語であるという間違った結論が証明できてしまう．したがって，背理法により，CFL は共通部分で閉じていないと結論できる．　▨

一方，共通部分演算に関してより弱い主張ができる．文脈自由言語は正則言語との共通部分で閉じている．形式的な記述と証明を次の定理で与える．

定理 7.27　L が CFL で R が正則言語ならば，$L \cap R$ は CFL である．

証明　CFL のプッシュダウンオートマトン (PDA) による表現と正則言語の有限オートマトン (FA) による表現とを使って，正則言語の共通部分を得るために二つの FA を "平行に" 走らせた定理 4.8 の証明を一般化する．ここでは FA と PDA とを平行に走らせる．その結果は，図 7.9 に示したように，再び PDA になる．

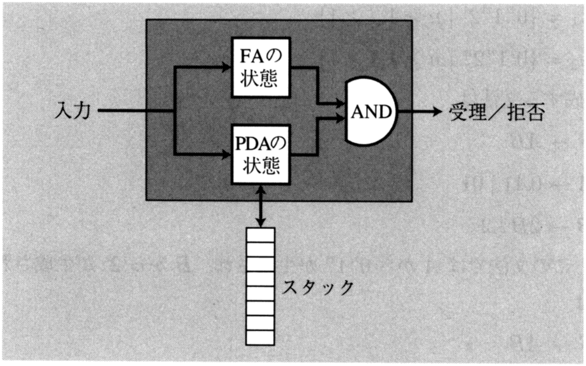


図 7.9　PDA と FA を平行に走らせて新しい PDA を作る

形式的には，

$$F = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

を L を受理する PDA とし，

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$$

を R を受理する DFA とする．以下のように PDA

$$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$$

を構成する．ここで，$\delta((q,p),a,X)$ は以下の条件を満たすすべての $((r,s),\gamma)$ の集合である．

(1)　$s = \hat{\delta}_A(p,a)$

(2)　$(r,\gamma) \in \delta_P(q,a,X)$

すなわち，PDA P' では PDA P の動作に加えて，状態の第 2 成分として，DFA の状態を持ち運ぶ．a は Σ の記号かも知れないし，$a = \varepsilon$ かも知れないことに注意しよ

う．前者の場合，$\hat{\delta}_A(p,a) = \delta_A(p,a)$ である．一方 $a = \varepsilon$ なら $\hat{\delta}_A(p,a) = p$ であり，PDA が ε 入力上で動いている間は A の状態を変えない．

PDA の動作数に関する簡単な帰納法で，$(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \gamma)$ は $((q_P, q_A), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((q,p), \varepsilon, \gamma)$ の必要十分条件であることが証明できる．ここで $p = \delta(q_A, w)$ である．帰納法による証明は，読者への演習問題としよう．(q,p) が P' の受理状態である必要十分条件は q が P の受理状態でかつ p が A の受理状態であることだから，P' が w を受理する必要十分条件は P と A がともに w を受理すること，すなわち $w \in L \cap R$ である．▨

例 7.28　図 6.6 で，C プログラム中の if や else の現れかたに関する規則の最小違反例 (違反する最初の場所まで) を受理する PDAP_F を設計した，if を i で，else を e で表した最小違反例の集合を L とする．受理状態で L を受理する PDA P_F は，次のように定義される．

$$P_F = (\{p,q,r\},\ \{i,e\},\ \{Z,X_0\},\ \delta_F,\ p,\ X_0,\ \{r\})$$

ただし δ_F は次の遷移規則から成る．

(1)　$\delta_F(p, \varepsilon, X_0) = \{(q, ZX_0)\}$.

(2)　$\delta_F(q, i, Z) = \{(q, ZZ)\}$.

(3)　$\delta_F(q, e, Z) = \{(q, \varepsilon)\}$.

(4)　$\delta_F(q, \varepsilon, X_0) = \{(r, \varepsilon)\}$.

次に，$\mathbf{i}^*\mathbf{e}^*$ を受理する有限オートマトン

$$A = (\{s,t\}, \{i,e\}, \delta_A, s, \{s,t\})$$

を構成する．すなわち A は i の列に続いて e の列が続くすべての列を受理する．この言語を R とする．遷移関数 δ_A は次の規則から成る．

(1)　$\delta_A(s, i) = s$.

(2)　$\delta_A(s, e) = t$.

(3)　$\delta_A(t, e) = t$.

厳密にいえば，A は定理 7.27 で仮定した DFA ではない．状態 t で入力 i の遷移が定義されていないように，死状態を省略している．しかし，構成する PDA は非決定性でもよいから，同じ構成法が NFA と PDA の対に対しても，うまく働く．今の場合，構成された PDA は，ある入力に対して "死ぬ" ことはあるにしても，実質的に決定性である．

次の PDA を構成する．

$$P = (\{p,q,r\} \times \{s,t\},\ \{i,e\},\ \{Z,X_0\},\ \delta,\ (p,s),\ X_0,\ \{r\} \times \{s,t\})$$

δ の遷移を以下に並べ上げる．遷移規則には，その規則を生じた PDA P_F の規則に応じて 1 から 4 までの番号を付け，DFA A の規則に応じて a から c までの記号を付けた．PDA P_F の ε-規則から作られる規則の場合，対応する A の規則はない．これらの規則は，"怠惰" なやり方で作られていることに注意せよ．すなわち，P_F と A の開始状態の対 (p,s) から始めて，P の到達可能な状態対だけに対して規則を構成している．

(1) $\delta((p,s),\varepsilon,X_0) = \{((q,s),ZX_0)\}$.

(2a) $\delta((q,s),i,Z) = \{((q,s),ZZ)\}$.

(3b) $\delta((q,s),e,Z) = \{((q,t),\varepsilon)\}$.

(4) $\delta((q,s),\varepsilon,X_0) = \{((r,s),\varepsilon)\}$.

注：この規則は，実は決して実行されない．実際，入力記号 e を見ずにスタックをポップすることは不可能で，かつ P は e を見ると直ちに状態の第 2 成分を t にしてしまう．

(3c) $\delta((q,t),e,Z) = \{((q,t),\varepsilon)\}$.

(4) $\delta((q,t),\varepsilon,X_0) = \{((r,t),\varepsilon)\}$.

言語 $L \cap R$ は，いくつかの i の後ろにそれより 1 個多い e が並んだ列の集合 $\{i^n e^{n+1} \mid n \geq 0\}$ である．この集合は "if" のかたまりの後に "else" のかたまりが続く列で，"if-else" の制限に違反する列である．これは明らかに CFL で，規則 $S \to iSe \mid e$ を持つ文法で生成される．

構成した PDA P は言語 $L \cap R$ を次のように受理する．まず Z をスタックに載せた後，入力 i に対して，状態を (q,s) にとどめたまま，さらに Z をスタックに積んでいく．P は入力 e を見ると直ちに，状態 (q,t) に遷移しスタックのポップを始める．P はスタックに X_0 が現れる前に i を読むと死ぬ．X_0 がスタックに現れると同時に，P は状態 (r,t) に遷移し，入力を受理する． ▨

CFL は CFL との共通部分演算では閉じていないが，正則集合との共通部分演算では閉じていることがわかったので，CFL の集合差や補集合演算に関しても同様のことが成り立つことがわかる．これらの結果を次の定理にまとめる．

定理 7.29 CFL L, L_1, L_2 および正則集合 R に対して以下が成り立つ．

(1)　$L-R$ は CFL である．

(2)　$\overline{L}$ は CFL とは限らない．

(3)　L_1-L_2 は CFL とは限らない．

証明　(1) $L-R=L\cap\overline{R}$ である．R が正則ならば定理 4.5 より $\overline{R}$ も正則である．よって定理 7.27 より，$L-R$ は CFL である．

(2) L が CFL なら $\overline{L}$ も常に CFL だとしてみよう．

$$L_1\cap L_2=\overline{\overline{L_1}\cup\overline{L_2}}$$

であり，CFL は集合和で閉じているから，CFL は共通部分で閉じていることになるが，例 7.26 から CFL は共通部分演算で閉じていない．

最後に (3) を証明する．すべてのアルファベット Σ に対して，Σ^* は CFL である．実際，正則言語 Σ^* を受理する PDA や文法を構成するのはたやすい．したがって，任意の CFL L_1,L_2 に対して，L_1-L_2 が CFL であるとすれば，CFL L に対して Σ^*-L も CFL になるはずである．しかし，アルファベット Σ を適切に選べば，$\Sigma^*-L=\overline{L}$ となり，(2) に矛盾する．よって背理法により，L_1-L_2 は CFL とは限らない．▨

7.3.5　逆準同型写像

4.2.4 項で扱った「逆準同型写像」を復習しておこう．h が準同型写像で L が言語とする．$h^{-1}(L)$ は，$h(w)\in L$ であるような列 w の集合である．正則言語が逆準同型写像で閉じていることの証明の概要を図 4.6 に示した．図 4.6 のオートマトンは，入力記号 a に対し，入力 $h(a)$ に対するもとのオートマトンの動作を模倣する．

CFL の逆準同型写像のもとでの閉包性は，有限オートマトンの代わりに PDA を使って，ほとんど同じように証明できる．しかし，PDA では，有限オートマトンでは生じなかった一つの問題に直面する．入力列上での有限オートマトンの一連の動作は，結果として一つの状態遷移になるので，構成されたオートマトンについてみると，一つの入力記号上で行われる一つの動作に見える．

PDA の場合，これと異なり，動作列は一つの入力記号上での動作には見えないかも知れない．特に，n 回の動作で PDA は最大 n 個の記号をスタックからポップできるが，1 回の動作でポップできる記号は 1 個だけである．したがって，PDA に対する図 4.6 と同様の構成法は，いくらか複雑になる．それを図 7.10 に示した．付け加えられた主要なアイデアは，入力 a が読まれると，$h(a)$ が「バッファ」に置かれることである．$h(a)$ の記号は 1 文字ずつ使われ，模倣

されるもとの PDA に与えられる．バッファが空になったときに，構成される PDA は次の入力記号を読み，それに準同型写像を適用してバッファに貯える．この構成法を次の定理の証明で形式的に述べよう．

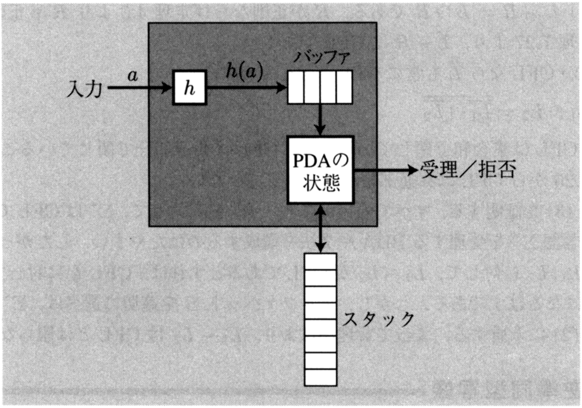


図 7.10　与えられた PDA が受理する言語の逆準同型写像の像を受理する PDA の構成

定理 7.30　L を CFL，h を準同型写像とする．$h^{-1}(L)$ は CFL である．

証明　h はアルファベット Σ の記号に適用され，T^* 中の列を生み出すものとする．さらに L はアルファベット T 上の言語とする．上で述べたように，L を最終状態で受理する PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$ から，新しい PDA

$$P' = (Q', \Sigma, \Gamma, \delta', (q_0, \varepsilon), Z_0, F \times \{\varepsilon\}) \tag{7.1}$$

を構成する．ここで，

(1)　Q' は次のような対 (q, x) の集合である．

(a)　q は Q に属す状態．

(b)　x はある $a \in \Sigma$ に対する列 $h(a)$ のある接尾列．

すなわち，P' の状態の第 1 成分は P の状態であり，第 2 成分はバッファである．バッファには定期的に列 $h(a)$ が貯えられ，記号が頭の方から模倣される PDA に使われて，縮んでいく．Σ は有限集合であり，$h(a)$ もすべて有限長であるから，P' の状態数も有限である．

(2)　δ' を以下で定義する．

(a)　すべての入力記号 $a \in \Sigma$，状態 $q \in Q$，スタック記号 $X \in \Gamma$ に対し，

$\delta'((q,\varepsilon),a,X) = \{((q,h(a)),X)\}$．ここで $a \neq \varepsilon$ である．バッファが空になると P' は次の入力記号 a を読み込んで，$h(a)$ をバッファに置くことができる．

(b) $(p,\gamma) \in \delta(q,b,X)$ ($b \in T$ または $b = \varepsilon$) ならば，$((p,x),\gamma) \in \delta'((q,bx),\varepsilon,X)$．すなわち，$P'$ は，バッファの最前部の記号を使って，P の動きを模倣をする．$b \in T$ ならばバッファは空でないが，$b = \varepsilon$ ならばバッファは空かもしれない．

(3) 式 (7.1) で定義したように，P' の開始状態は (q_0,ε) である．すなわち，P' は P の開始状態と空のバッファで出発する．

(4) P' の最終状態は式 (7.1) より，P の最終状態 q と空のバッファを表す ε の対である．

次の主張は，P' と P の関係を表す．

- $(q_0,h(w),Z_0) \overset{*}{\underset{P}{\vdash}} (p,\varepsilon,\gamma)$ の必要十分条件は $((q_0,\varepsilon),w,Z_0) \overset{*}{\underset{P'}{\vdash}} ((p,\varepsilon),\varepsilon,\gamma)$ である．

証明は両方向とも，それぞれの PDA の動作数に関する帰納法で証明される．十分性の証明では，バッファが空でなくなると P' はもう新たに入力記号を読むことができず，バッファが再び空になるまで P を模倣することしかできないことに着目する (バッファが空になっても P の模倣を続けることはできる)．詳細は演習問題として読者にまかせよう．

P' と P の動作に関する上記の関係を認めてしまえば，P' の最終状態の定義から，P が $h(w)$ を受理する必要十分条件は P' が w を受理することである．したがって，$L(P') = h^{-1}(L(P))$ である．▨

7.3.6　7.3 節の演習問題

♣ **問 7.3.1**　CFL は次の演算で閉じていることを示せ．

*a) 問 4.2.6(c) で定義した *init*．ヒント：言語 L を生成する CNF から始めよ．

*!b) 問 4.2.2 で定義した演算 L/a．ヒント：再び L の CNF から始めよ．

!!c) 問 4.2.11 で定義した *cycle*．ヒント：PDA に基づく構成法を試せ．

♣ **問 7.3.2**　次の言語を考える．

$$L_1 = \{a^n b^{2n} c^m \mid n,m \geq 0\}$$

$$L_2 = \{a^n b^m c^{2m} \mid n,m \geq 0\}$$

a) これらの言語がそれぞれ CFL であることを，それぞれを生成する文法を与えることによって示せ．

!b) $L_1 \cap L_2$ は CFL か．その答を証明せよ．

♣!! **問 7.3.3**　CFL は次の演算で閉じていないことを示せ．

*a)　問 4.2.6(a) で定義した *min*

b)　問 4.2.6(b) で定義した *max*

c)　問 4.2.8 で定義した *half*

d)　問 4.2.7 で定義した *alt*

♣　**問 7.3.4**　二つの列 w と x の**シャッフル** (shuffle) とは，それらを切り混ぜて得られる語の全体の成す集合である．正確には，$\mathit{shuffle}(w,x)$ は次の条件を満たす列 z の集合である．

(1)　z 上の文字の出現は，w または x のどちらか一方だけに割り当てられる．

(2)　w に割り当てられた z 上の出現を左から右に並べると w になる．

(3)　x に割り当てられた z 上の出現を左から右に並べると x になる．

例えば $w = 01, x = 001$ とすると，$\mathit{shuffle}(01, 110)$ は，列の集合
$\{01110, 01101, 10110, 10101, 11010, 11001\}$ である．

これらの列が $\mathit{shuffle}(01, 110)$ に属すことを説明するために，4 番目の列 10101 を考えよう．2 番目と 5 番目の文字を並べれば，01 になり，残りの 1,3,4 番目の文字を並べると 110 になる．最初の列 01110 では，1 番目と 2,3,4 番目のどれかの文字を合わせれば 01 が得られ，残りの 3 文字をもとの順に並べれば，110 になる．言語の**シャッフル**も定義できる．すなわち，$\mathit{shuffle}(L_1, L_2)$ は，L_1 の列 w と L_2 の列 x のすべての対 (w, x) に対して，集合 $\mathit{shuffle}(w, x)$ の和集合をとったものである．

a)　$\mathit{shuffle}(00, 111)$ を求めよ．

*b)　$L_1 = L(\mathbf{0}^*), L_2 = \{0^n1^n \mid n \geq 0\}$ に対し $\mathit{shuffle}(L_1, L_2)$ を求めよ．

*!c)　L_1, L_2 が正則言語のとき，$\mathit{shuffle}(L_1, L_2)$ も正則言語であることを示せ．**ヒント**：L_1, L_2 の DFA を使え．

!d)　L が CFL で R が正則言語のとき，$\mathit{shuffle}(L, R)$ は CFL であることを示せ．**ヒント**：L の PDA と R の DFA を使え．

!!e)　L_1, L_2 がともに CFL のとき，$\mathit{shuffle}(L_1, L_2)$ は CFL とは限らないことを示す実例を与えよ．

♣*!! **問 7.3.5**　列 y が列 x の**置換** (permutation) であるというのは，y 上の文字を並べ替えて，x を作れるときである．たとえば，011 の置換は 110, 101, 011 の三つである．L を言語とする．$\mathit{perm}(L)$ は L に属す列の置換の全体である．たとえば，$L = \{0^n1^n \mid n \geq 0\}$ のとき，$\mathit{perm}(L)$ は同数の 0 と 1 から成る列全体の集合である．

a)　アルファベット $\{0, 1\}$ 上の正則言語 L で，$\mathit{perm}(L)$ が正則言語でない例を示し，それを証明せよ．**ヒント**：正則言語でその置換が，同数の 0 と

1から成る列の集合になるものを探せ．

b) アルファベット $\{0,1,2\}$ 上の正則言語 L で，$perm(L)$ が CFL でない例を示せ．

c) 2文字から成るアルファベット上の正則言語 L の置換 $perm(L)$ は，CFL であることを示せ．

♣ **問 7.3.6**　定理 7.25：「CFL は逆順で閉じている」の形式的証明を与えよ．

♣ **問 7.3.7**　定理 7.27 の証明を完成させよ．すなわち，

$$(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \gamma)$$

の必要十分条件は $((q_P, q_A), w, Z_0) \overset{*}{\underset{P}{\vdash}} ((q,p), \varepsilon, \gamma)$ かつ $p = \delta(p_A, w)$ であることを証明せよ．

7.4　文脈自由言語の決定問題

文脈自由言語に関するどのような決定問題に答えることができるか，調べてみよう．正則言語の決定問題を調べた 4.3 節にならって，問題の出発点は CFL を表現する CFG または PDA である．6.3 節より，CFG と PDA とは互いに変換可能なことがわかっているので，どちらでもより便利な方の表現が与えられていると仮定してよい．

CFL については，決定可能な問題が少ない．決定可能な代表的な問題は，言語が空か否かと，与えられた列がその言語に属すか否かとである．この節の最後では，後に第 9 章で，決定不能，すなわち，それを解くアルゴリズムがないことを示す問題について簡単に議論する．この節では，最初に，言語の CFG と PDA による記述の間の変換の複雑さについて調べる．これらの変換は，与えられた表現から CFL の性質をどの位効率的に決定できるかという問題に関連する．

7.4.1　CFG と PDA の間の変換の複雑さ

CFL に関する問題の決定アルゴリズムを議論する前に，CFL の一つの表現から別の表現への変換の複雑さを考えよう．変換の計算時間は，言語が，決定アルゴリズムが仮定しているのとは異なる表現形式で与えられた場合の決定アルゴリズムのコストの一部である．

以下では，n を PDA または CFG の表現全体の長さとする．この変数を文法やオートマトンの大きさの表現として使うことはある意味で“荒っぽく”て，

アルゴリズムの中には，文法中の変数の個数や PDA の遷移関数中に現れるスタック列の長さの和，といったより細かな指標を用いて，その計算時間をより正確に記述できるものがある．

しかしながら，表現全体の長さという尺度は，もっとも重要な論点を区別するのに十分である．それは，アルゴリズムが長さに比例するのか (つまり，その入力を読むのとほとんど変わらない時間か)，長さの指数関数か (つまり，ごく小さな実例でしか変換を実行できないか)，あるいは 2 次以上の多項式か (つまり，大きな実例に対しても実行できるが時としてその実行時間はかなり大きくなるか)，を区別できる．

今まで見てきた変換の中には入力の大きさに比例するものがいくつかある．線形時間であることから，それらが出力する表現は，すばやく得られるだけでなく入力と同程度の大きさである．そのような変換には

(1)　定理 6.13 のアルゴリズムによる CFG から PDA への変換．

(2)　定理 6.9 の構成法を使った空スタック受理 PDA から最終状態受理 PDA への変換．

(3)　定理 6.11 の構成法を使った，最終状態受理 PDA から空スタック受理 PDA への変換．

がある．一方，PDA から文法への変換 (定理 6.14) はずっと複雑である．まず，入力の全体長 n は状態やスタック記号の数の上限でもあるから，文法のために構成される $[pXq]$ の形の変数は n^3 個を越えない．しかし，PDA に多数の記号をスタックに置く遷移があると，変換の計算時間はその指数関数になる可能性がある．そして一つの規則がスタックにほぼ n 個の記号を置くこともあり得るので，計算時間が n の指数関数になることも考えられる．

実際，遷移規則 $(r_0, Y_1Y_2\cdots Y_k) \in \delta(q, a, x)$ からの文法規則の構成を見直すと，それが $[qXr_k] \to [r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k]$ の形の規則を，状態のあらゆる列 $r_1, r_2, \ldots, r_k$ に対して生じさせることがわかる．k は n に近い値をとり得るし，状態数も n に近くなり得るから，これらの規則の総数は n^n にもなり得る．これでは，PDA が長いスタック列を書き込む規則をただ一つ持つだけで，通常の大きさの PDA からの構成も遂行できない．

幸運にも，この最悪の場合を避けることができる．問 6.2.8 で示したように，長さ k のスタック記号列のプッシュを，それぞれは 1 記号をプッシュする k 個の

動作列に分解することができる．すなわち，$(r_0, Y_1Y_2\cdots Y_k) \in \delta(q,a,x)$ に対し，新しい状態 $p_2, p_3, \ldots, p_{k-1}$ を導入し，$\delta(q,a,x)$ 中の $(r_0, Y_1Y_2\cdots Y_k)$ を $(p_{k-1}, Y_{k-1}Y_k)$ で置き換え，新しい遷移，

$$\delta(p_{k-1}, Y_{k-1}) = \{(p_{k-2}, Y_{k-2}Y_{k-1})\},$$

$$\delta(p_{k-2}, Y_{k-2}) = \{(p_{k-3}, Y_{k-3}Y_{k-2})\},$$

$$\ldots,$$

$$\delta(p_2, Y_2) = \{(r_0, Y_1Y_2)\}$$

を導入する．

こうすれば，どの遷移も長さ 3 以上のスタック記号を持たない．高々n 個の新しい状態が追加され，遷移規則の記述の全長は高々定数倍にすぎず，$O(n)$ のままである．$O(n)$ 個の遷移があり，それぞれの遷移から，文法規則を作るのに必要な変数は 2 個以下なので，$O(n^2)$ の文法規則しか生じない．したがって，構成された文法の長さは $O(n^3)$ であり，3 乗の時間で構成できる．この (非形式的な) 議論の結果を以下の定理にまとめよう．

定理 7.31　長さ n の記述を持つ PDA P から，高々$O(n^3)$ の長さの CFG G を生成する $O(n^3)$ 時間アルゴリズムが存在する．G は P が空スタックで受理するのと同じ言語を生成する．また，G は P が最終状態で受理する言語を生成するようにもできる．　▨

7.4.2　チョムスキー標準形への変換の計算時間

決定アルゴリズムは，最初 CFG をチョムスキー標準形にすることを必要とするかもしれないので，任意の文法を CNF 文法に変換するのに使ったいくつかのアルゴリズムの計算時間も調べるべきである．ステップの多くは，文法の記述の長さを，定数倍を除いて保存する．すなわち，長さ n の文法から，長さ $O(n)$ の文法を作る．これらの好ましい結果は，次の一連の観察にまとめられる．

(1)　適切なアルゴリズム (7.4.3 項を見よ) を用いることにより，文法の到達可能かつ生成的な記号を見つけることは $O(n)$ 時間でできる．したがって，無用な記号の除去は $O(n)$ 時間ででき，文法の大きさは増えない．

(2)　単位対を構成し，単位規則を除去することは，7.1.4 項で示したように $O(n^2)$ 時間かかり，その結果得られる文法の記述は長さ $O(n^2)$ になる．

(3) 規則の本体に現れる終端記号を変数に置き換えるのは，7.1.5 項 (チョムスキー標準形) で示したように，$O(n)$ 時間かかり，長さ $O(n)$ の文法が得られる結果となる．

(4) 長さ 3 以上の本体を持つ規則を，本体の長さが 2 の規則に分解することは，7.1.5 項でやったように，$O(n)$ 時間かかり，長さ $O(n)$ の文法が得られる．

7.1.3 項の ε-規則の除去には注意が必要である．本体の長さが k の規則があると，その規則から $2^k - 1$ 個の規則が新しい文法に加わるかも知れない．k は n に比例する可能性があるので，構成のこの部分は，$O(2^n)$ 時間かかり得るし，長さ $O(2^n)$ の文法ができてしまうかも知れない．

この指数的爆発を避けるためには，生成規則の本体の長さを制限すればよい．チョムスキー標準形を構成した 7.1.5 項の技巧は，終端記号を持たないものだけでなく，どんな規則の本体にも適用できる．したがって，ε-規則を除去するまえの準備段階として，本体が長さ 3 以上の規則を本体が長さ 2 の規則列に分割することを薦める．このステップは $O(n)$ 時間かかり，得られる文法の長さも線形でしか増えない．しかし，ε-規則を除去する 7.1.3 項の構成法は，規則の本体の長さが 2 以下のとき，計算時間は $O(n)$ で，結果の文法も長さ $O(n)$ である．

この変更によって，CNF の構成の全体で，線形でないステップは単位規則の除去の $O(n^2)$ だけであり，次の結果を得る．

定理 7.32 与えられた長さ n の文法 G に対して，G と等価な CNF 文法を $O(n^2)$ で見つけることができ，結果の文法は長さ $O(n^2)$ である． ▨

7.4.3 CFL の空集合検査

CFL L が空か否かを決定するアルゴリズムはすでに見てきた．言語 L の文法が与えられたとき，7.1.2 項のアルゴリズムを使って G の開始記号 S が生成的か否か，すなわち一つでも列を生成するか否かを決定できる．$L = \varnothing$ の必要十分条件は S が生成的でないことである．

この検査は重要なので，文法 G のすべての生成的記号を見つけるのにかかる時間の見積もりを詳しく調べよう．G の (記述の) 長さを n とする．G には n のオーダーの個数の変数があり得る．そして生成的記号の帰納的発見手順の各

ステップは，G のすべての規則を調べるのに $O(n)$ の時間がかかる．各ステップで 1 個ずつ新しい生成的変数が見つかったとすると，$O(n)$ 回のステップが必要になる．したがって，生成的記号の検査は，素朴に実現すると $O(n^2)$ 時間かかる．

しかし，より巧妙なアルゴリズムがあり，前もってデータ構造をうまく組み立てることにより，生成的記号の発見を $O(n)$ 時間で達成できる．図 7.11 に示したデータ構造は，図の一番左にある配列を持つ．配列は変数を添え字に持ち，変数の生成能力を確定したか否かを示す欄を持つ．図 7.11 では，B が生成的であると判明し，A はまだ未定の状況を表している．アルゴリズムが生成的であると判断しなかった変数は，実際に生成的ではないので，最後に疑問符 “?” は “no” に変わる．

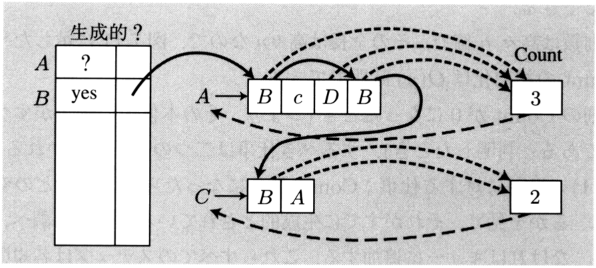


図 7.11　線形時間空集合検査のためのデータ構造

生成規則を前処理して，何種類かの有用なリンクを組み立てる．最初に，各変数に対して，その変数の (規則本体中の) 出現をリンクでつなぐ．たとえば，変数 B のリンクを実線で示した．各規則に対して，本体における生成能力がまだ決定されていない変数の出現数を保持する Count がある．破線は各出現から Count へのリンクを示す．図 7.11 は，B が生成的であることがちょうどわかったところで，まだどの変数も Count の計算に入れられていない状況を示している．

B が生成的であると判明したとしよう．B の出現リンクをたどり，規則の Count を，出現ごとに 1 減らす．Count は規則頭部の変数の生成能力を決定するために使われる．Count が 0 になれば，頭部の変数は生成的である．点線で

線形時間空集合検査アルゴリズムの他の使用法

7.4.3 項で変数の生成能力検査に使ったデータ構造と個数カウント技法は，7.1 節の他の性質の線形時間検査に使用できる．二つの重要な例は次のものである．

(1)　どの記号が到達可能か?

(2)　どの記号が消去可能か?

示されたリンクからその変数を確定し，そこから得られる結論を (今 B でやったように) 調べるために，(図には示されていない) 生成的変数のキューに加える．

このアルゴリズムが $O(n)$ 時間であることを示さなければならない．主要な論点は以下の通りである．

- 大きさ n の文法の変数は高々n 個なので，配列の作成と初期化は $O(n)$ 時間でできる．
- 規則数は高々 n 個で，その全長は高々n なので，図 7.11 に示したリンクと Count の初期化は $O(n)$ 時間でできる．
- 規則の Count が 0 になったとき (つまり，その本体中の変数がすべて生成的であると判明したとき)，するべき仕事は二つの範疇に分かれる．
  (1)　規則に対する仕事：Count が 0 になったら，頭部にどの変数があるかを見て，それがすでに生成的とされているかどうか調べ，そうでなければキューに追加する．これらすべてのステップは各規則に対して $O(1)$ で，このタイプの仕事は全体として高々$O(n)$ 時間でできる．
  (2)　変数 A の (規則本体中の) 各出現位置を訪れる仕事：この仕事は A の総出現数に比例する．それゆえ，すべての生成的記号を処理する時間の総計は，規則の本体の長さの総和に比例し，$O(n)$ である．

したがってこのアルゴリズムの総計算時間は $O(n)$ である．

7.4.4 CFL への所属検査

CFL L に列 w が所属するか否かは決定可能である．言語 L が文法か PDA で与えられ，その大きさは w とは無関係な定数と考えて，$|w|$ の指数時間かかるような，非効率的な決定方法がいくつかある．たとえば，L の与えられた表現 (それが何であっても) を CNF 文法に変換するところから始めよう．チョムスキー標準形文法の導出木は 2 分木であるから，w の長さが n なら，その導出

木の中の変数ラベルを持つ節点はちょうど $2n-1$ 個である (この結果は簡単な帰納的証明を持ち，証明は読者に任せる)．可能な木と節点のラベル付けの総数はある定数の n 乗に "すぎない"．それゆえ，原理的には，それらすべてを並べ上げ，どれか実際に w を生成するものがあるか，調べることができる．

これよりずっと効率のよい技法で,「ダイナミック・プログラミング」あるいは,「表埋めアルゴリズム」として知られるものが存在する．このアルゴリズムは CYK アルゴリズム[3]と呼ばれ，L の CNF 文法 $G=(V,T,P,S)$ から出発する．このアルゴリズムは，入力列 $w=a_1a_2\cdots a_n\in T^*$ に対し，$O(n^3)$ 時間で，$w\in L$ か否かを明らかにする表を作成する．計算時間は w の長さで測られる．文法自身は固定されていて，その大きさは，計算時間の定数因子にしか関わらないことに注意せよ．

CYK アルゴリズムでは，図 7.12 に示したような三角形の表を作成する．水平軸は列 $w=a_1a_2\cdots a_n$ の各出現に対応し，図では長さ $n=5$ と仮定している．表の要素 X_{ij} は $A\overset{*}{\Rightarrow}a_ia_{i+1}\cdots a_j$ となる変数 A の集合である．我々が問題にするのは，S が X_{1n} に属すか否かである．なぜならそれが $S\overset{*}{\Rightarrow}w$，すなわち，$w\in L$ と同値だから．

表は，行ごとに下から上に埋めていく．各行は，部分列の長さに対応し，一番下の行にある列は長さ 1 の部分列，下から 2 番目の行にある列は長さ 2 の部分列等々と続き，一番上にある列が長さ n の部分列，つまり w 自身に対応して

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

図 7.12 CYK アルゴリズムで構成される表

[3] この名前は本質的に同じアイデアを独立に発見した 3 人，クック (J. Cocke)，ヤンガー (D. Younger)，カサミ (T. Kasami) に由来する．

いる．次に述べる手法によって，表の 1 要素は $O(n)$ で計算できる．表の要素は，$n(n+1)/2$ 個なので，表作成全体には $O(n^3)$ 時間かかる．以下は，X_{ij} を計算する帰納的アルゴリズムである．

基礎：最初の行を以下のように計算する．位置 i で始まり i で終わる部分列は終端記号 a_i そのものであり，文法は CNF であるから，列 a_i を導出する方法は，$A \to a_i$ という形の規則を使うことだけである．したがって，X_{ii} は $A \to a_i$ という規則が G にあるような変数 A の集合である．

帰納：X_{ij} を計算したいとしよう．X_{ij} は行 $j-i+1$ にあり，その行より下にあるすべての要素はすでに計算済みであるとする．すなわち，$a_i a_{i+1} \cdots a_j$ より短い部分列，特に $a_i a_{i+1} \cdots a_j$ の真の接頭列や接尾列については，それらを生成する変数がわかっているものとする．$j-i>0$ が仮定されている ($i=j$ の場合は基礎ステップ) ので，任意の導出 $A \stackrel{*}{\Rightarrow} a_i a_{i+1} \cdots a_j$ はある導出 $A \Rightarrow BC$ から始まる．B は $a_i a_{i+1} \cdots a_j$ のある接頭列を導出する．$B \stackrel{*}{\Rightarrow} a_i a_{i+1} \cdots a_k$ ($j < k$) としよう．残りの $a_{k+1} a_{k+2} \cdots a_j$ は C から導出される，すなわち，$C \stackrel{*}{\Rightarrow} a_{k+1} a_{k+2} \cdots a_j$ である．

結局，A を X_{ij} に入れるためには，ある整数 k と次を満たす変数 B, C とを見つけなければならない．

(1)　$i \le k < j$

(2)　$B \in X_{ik}$

(3)　$C \in X_{k+1,j}$

(4)　$A \to BC$ が G の規則

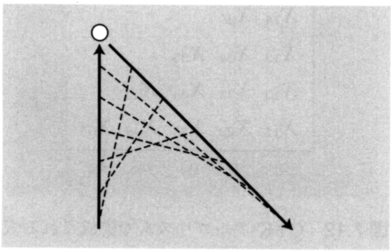


図 7.13　X_{ij} の計算には，真下から上がる列と右下がりの対角線にそった列とを照合させる

上の条件を満たす A を見つけるには，高々 n 個の，すでに計算された集合の対 $(X_{ii}, X_{i+1,j})$，$(X_{i,i+1}, X_{i+2,j})$，$\ldots$，$(X_{i,j-1}, X_{j,j})$ を照合すればよい．対の第 1 項と第 2 項のなすパターンは，図 7.13 に示すように，X_{ij} の真下の列を第 1 項が上ってくるのと同時に，第 2 項が対角線上を下りていく．

定理 7.33　上に述べたアルゴリズムはすべての i, j $(i \le j)$ に対して，X_{ij} を正しく計算する．したがって，$w \in L(G)$ の必要十分条件は $S \in X_{1n}$ である．また，このアルゴリズムの計算時間は $O(n^3)$ である．

証明　このアルゴリズムが変数の正しい集合を見つける理由は，アルゴリズムの基礎ステップと帰納ステップを導入したときに説明した．計算時間に関してはいえば，計算すべき要素が $O(n^2)$ 個あり，それぞれが，要素の高々 n 個の対を照合・計算する必要がある．ここで，各集合 X_{ij} には多くの変数が含まれ得るけれども，文法 G は固定されており変数の個数は，所属性の検査対象である列 w の長さ n に依存せず一定であることを思い出そう．したがって，二つの表要素 $X_{ik}, X_{k+1,j}$ を照合し，X_{ij} に入れる変数を見つけるための計算時間は $O(1)$ である．X_{ij} に対して，そのような対は高々n 個なので，全体の計算量は $O(n^3)$ である．　▨

例 7.34　CNF 文法 G

$$
\begin{aligned}
S &\rightarrow AB \mid BC \\
A &\rightarrow BA \mid a \\
B &\rightarrow CC \mid b \\
C &\rightarrow AB \mid a
\end{aligned}
$$

に対し，$baaba$ の $L(G)$ への所属問題を調べよう．図 7.14 はこの列に対して作られた表を示す．

最初 (一番下) の行を組み立てるために，基礎ステップの規則を使う．それには，どの変数が規則の本体 a を持ち (そのような変数は A, C)，どの変数が規則の本体 b を持つか (B のみ)，を調べるだけでよい．したがって，上の図で a が書かれた位置には要素 $\{A, C\}$ があり，b が書かれた位置には要素 $\{B\}$ がある．すなわち，$X_{11} = X_{44} = \{B\}$ であり，$X_{22} = X_{33} = X_{55} = \{A, C\}$ である．

2 行目には，$X_{12}, X_{23}, X_{34}, X_{45}$ の値が並んでいる．例として，どのように X_{12} が計算されるか見てみよう．位置 1 から 2 までの列 ab を二つの空でない部

$\{S,A,C\}$				
-	$\{S,A,C\}$			
-	$\{B\}$	$\{B\}$		
$\{S,A\}$	$\{B\}$	$\{S,C\}$	$\{S,A\}$	
$\{B\}$	$\{A,C\}$	$\{A,C\}$	$\{B\}$	$\{A,C\}$
b	a	a	b	a

図 7.14　CYK アルゴリズムによって，$baaba$ に対して構成された表

分列に分割する方法は 1 通りしかない．最初は位置 1 であり，2 番目は位置 2 である．変数が ba を生成するためには，規則の本体の最初の変数が $X_{11}=\{B\}$ に属し (すなわち a を生成し)，2 番目の変数が $X_{22}=\{A,C\}$ に属す (すなわち b を生成する) ような規則を持たなければならない．このような本体は BA, BC しかなく，文法を調べれば，規則 $A \to BA$ と $S \to BC$ だけがこれらの本体を持つ規則であるとわかる．したがって，これらの規則の頭部 A と S が X_{12} を構成する．

より複雑な例として，X_{24} の計算を考えよう．位置 2 から 4 までを占める列 aab の分割は，最初の部分列が位置 2 で終わる場合と位置 3 で終わる場合とがある．すなわち，X_{24} の定義では，$k=2$ または $k=3$ を選べる．したがって，集合 $X_{22}X_{34} \cup X_{23}X_{44}$ に属す規則の本体をすべて考えなければならない．この変数列の集合は，$\{A,C\}\{S,C\} \cup \{B\}\{B\} = \{AS, AC, CS, CC, BB\}$ である．この集合に属する五つの列のうち，規則の本体になっているのは CC だけであり，その頭部は B である．したがって，$X_{24}=\{B\}$ である．　▨

7.4.5　決定不能な CFL 問題のあらまし

次章では，コンピュータ上で動くどのようなアルゴリズムにも解くことのできない問題の存在を形式的に証明できる，注目すべき理論を展開する．その理論を使って，CFG や CFL に関連し，簡単に述べることのできる多くの問題が，それを解くアルゴリズムを持たないことが示される．しかし，ここでは文脈自由文法や言語に関する最も重要な解決不能問題のリストを挙げることに甘んじなければならない．以下の問題は決定不能である．

(1)　与えられた CFG はあいまいか．

(2)　与えられた CFL は本質的にあいまいか．

(3)　二つの CFL の共通部分は空集合か．

(4)　二つの CFL が等しいか．

(5)　与えられた CFL が Σ^* に等しいか．ここで Σ はその CFL のアルファベットである．

あいまい性に関する問題 (1) の，他とは少し異なる趣に注目しよう．(1) は文法に関する問題で，言語に関する問題ではない．他の問題はすべて，言語が文法や PDA で与えられるとしても，あくまでそれらが定める言語に関するものである．たとえば，(1) と異なり，問題 (2) は与えられた CFG G (または PDA) に対して，それと等価で，あいまいでない文法 G' があるかないかを問題にしている．G 自身があいまいでなければ，答は確かに "NO" である．しかし，G があいまいであっても，例 5.27 の数式の文法で学んだように，まだ同じ言語を生成する文法 G' であいまいでないものがある可能性がある．

7.4.6　7.4 節の演習問題

♣　**問 7.4.1**　以下を決定するアルゴリズムを示せ．

- *a)　与えられた文法 G の言語 $L(G)$ が有限集合か否か．
  ヒント：反復補題を使え．
- !b)　与えられた文法 G の言語 $L(G)$ が少なくとも 100 個の列を含むか否か．
- !!c)　与えられた文法 G とその変数 A に対して，A から始まる文形式が存在するか否か．
  ヒント：A が最初は文形式の途中に現れたとしても，導出の過程で，A の左にあった記号がすべて ε になってしまう可能性のあることを思い出せ．

♣　**問 7.4.2**　7.4.3 項に述べた技法を使い，CFG の次の問題に対する線形時間アルゴリズムを開発せよ．

- a)　どの記号が文形式中に現れるか．
- b)　どの記号が空列 ε を導出するか．

♣　**問 7.4.3**　例 7.34 の文法 G と，CYK アルゴリズムを使い，以下の列のどれが $L(G)$ に属するか決定せよ．

- *a)　$ababa$
- b)　$baaba$
- c)　$aabab$

♣ ＊**問 7.4.4**　任意の CNF 文法 G において，長さ n の列の導出木は，$2n-1$ 個の内点を持つ (すなわち，変数ラベルを持つ点が $2n-1$ 個ある) ことを示せ．

♣ ！**問 7.4.5**　CYK アルゴリズムを，与えられた入力に対して，その言語に属すか否かだけでなく，異なる導出木の個数を報告するように修正せよ．

7.5 第 7 章の要約

◆**無用な記号の除去**：CFG の記号は，終端記号列を何も生成しなかったり，開始記号から導出されるどの列にも現れなかったりする場合には除去できる．そのような無用な記号を除去するために，まず，変数が終端記号列を生成するか否か調べ，何も生成しない変数をそれが出現する規則とともに消去する．その後で，開始記号から到達可能でない記号を消去する．

◆**ε-規則と単位規則の除去**：空列以外の列を少なくとも一つ生成する CFG に対し，それと空列 ε を除いて等しい言語を生成する CFG で，ε-規則 (本体が ε の規則) や，単位規則 (本体が変数 1 個だけの規則) を持たないものが作れる．

◆**チョムスキー標準形**：空列以外の列を少なくとも一つ生成する CFG に対し，それと空列 ε を除いて等しい言語を生成する CFG で，チョムスキー標準形のもの，すなわち無用な記号を持たず，規則の本体が変数 2 個の列か終端記号かのどちらかであるものが存在する．

◆**反復補題**：任意の CFL において，その言語に属する十分長い列に対して，その両端の部分列を同時並行的に “反復” できるような部分列を見いだすことができる．すなわち，その両端を任意の同じ回数だけ繰り返しても，もとの言語に属する．反復される部分列がともに ε になることはない．この補題とそして問 7.2.3 で扱ったオグデンの補題と呼ばれるより強い補題とを使って，多くの言語が文脈自由言語でないことを示すことができる．

◆**文脈自由言語を保存する演算**：CFL は代入，集合和，連接，閉包 (*)，逆順，逆順同型の演算で閉じている．CFL は，共通部分，補集合の演算で閉じていないが，CFL と正則集合の共通部分はつねに CFL である．

◆**CFL の空集合性の検査**：与えられた CFG に対し，それが何らかの列を生成するか否かを決定するアルゴリズムが存在する．巧みに実現すれば，この検査は与えられた文法の大きさに比例する時間で行える．

◆**CFL の所属性検査**：クック-ヤンガー-カサミアルゴリズムは，与えられた列が与えられた文脈自由言語に所属するか否かを決定する．この検査には，n を調べる列の長さとして，$O(n^3)$ 時間かかる．

7.6　第 7 章の参考文献

チョムスキー標準形は [2] による．グライバック (Greibach) 標準形は [4] にあるが，問 7.1.11 であらましを述べた構成法は M. C. Paull による．

文脈自由言語の基本的な性質の多くは [1] による．これらのアイデアは反復補題，基本的閉包性，CFL の空集合性や有限性などの単純な問題の検査，を含む．さらに，[6] は共通部分と補集合のもとでの非閉包性の出典であり，[3] は CFL の逆準同型写像を含む補足的な閉包性を与えている．オグデンの補題は [5] による．

CYK アルゴリズムには，三つの知られた出所がある．J. Cocke の仕事は，私的に配布され出版されなかった．T. Kasami の本質的に同じアルゴリズムの翻訳は，米空軍の内部メモとして出されただけである．しかし，D. Younger の仕事は型通りに出版された [7]．

[1] Y. Bar-Hillel, M. Perles, and E. Shamir, "On formal properties of simple phrase-structure grammars," *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14** (1961), pp.143–172.

[2] N. Chomsky, "On certain formal properties of grammars," *Information and Control* **2**:2 (1959), pp.137–167.

[3] S. Ginsburg and G. Rose, "Operations which preserve definability in languages," *J.ACM* **10**:2 (1963), pp.175–195.

[4] S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars," *J.ACM* **12**:1 (1965), pp.42–52.

[5] W. Ogden, "A helpful result for proving inherent ambiguity," *Mathematical Systems Theory* **2**:3 (1969), pp.31–42.

[6] S. Scheinberg, "Note on the boolean properties of context-free languages," *Information and Control* **3**:4 (1960), pp.372–375.

[7] D. H. Younger, "Recognition and parsing of context-free languages in time n^3," *Information and Control* **10**:2 (1967), pp.189–208.

索　引

あ 行

か 行

さ 行

た　行

な 行

は 行

ま 行

著者略歴

野崎昭弘（のざき あきひろ）

1959年　東京大学理学部数学科卒業
1961年　東京大学大学院修士課程修了
現　在　大妻女子大学名誉教授
理学博士

高橋正子（たかはし まさこ）

1962年　お茶の水女子大学理学部数学科卒業
1972年　ペンシルバニア大学大学院博士課程修了
現　在　東京工業大学名誉教授
Ph.D.(ペンシルバニア大学)

町田　元（まちだ はじめ）

1970年　東京大学理学部数学科卒業
1974年　東京大学大学院修士課程修了
現　在　一橋大学名誉教授　博士(数理科学)

山崎秀記（やまざき ひでき）

1972年　東京大学教養学部基礎科学科卒業
1975年　東京大学大学院修士課程修了
現　在　一橋大学教授　理学博士

Information & Computing=3

オートマトン 言語理論 計算論 I [第2版]

1984年 8月25日 ©	初　版　発　行
2002年 3月10日	初版第21刷発行
2003年 4月10日 ©	第2版第1刷発行
2012年 2月10日	第2版第9刷発行

原著者　J. ホップクロフト
　　　　R. モトワニ
　　　　J. ウルマン
訳　者　野崎昭弘
　　　　高橋正子
　　　　町田　元
　　　　山崎秀記

発行者　木下敏孝
印刷者　杉井康之
製本者　関川安博

発行所　**株式会社　サ イ エ ン ス 社**

〒151–0051　東京都渋谷区千駄ヶ谷1丁目3番25号
営業☎(03) 5474–8500 (代) 振替 00170–7–2387
編集☎(03) 5474–8600 (代) FAX ☎(03) 5474–8900

印刷　(株) ディグ　　　　製本　(株) 関川製本所

ISBN4–7819–1026–2

PRINTED IN JAPAN